Programando com PASCAL

A Linguagem do Turbo Pascal e do Delphi

2ª Edição
Revisada

Aprenda Praticando

Mais de 90 exercícios para utilizar em treinamentos

Jaime Evaristo

Programando com Pascal

Jaime Evaristo

Professor Adjunto do Instituto de Computação da Universidade Federal de Alagoas

Prefácio à Segunda Edição

Embora com novo título, este livro é uma reedição do livro *Aprendendo a Programar Programando na Linguagem Pascal*, editado pela Editora Book Express no ano 2002.

Como foi dito no seu prefácio, a primeira edição deste livro foi uma reedição livre do livro *Aprendendo a Programar Programando em Turbo Pascal* lançado pela Editora da Universidade Federal de Alagoas (EDUFAL), em 1996, que, apesar das dificuldades de distribuição de uma editora universitária, teve sua edição esgotada em meados do ano de 2001.

Também como foi dito no seu prefácio, a primeira edição deste livro foi o terceiro livro da série *Aprendendo a Programar*, cujos dois primeiros foram *Aprendendo a Programar numa Linguagem Algorítmica Executável(ILA)* e *Aprendendo a Programar Programando em Linguagem C*. Sendo o terceiro livro de uma série, muitos dos problemas inerentes às primeiras edições já haviam sido resolvidos.

Os fatos expostos acima e o fato de que não recebi mensagens de leitores com qualquer tipo de reprovação ao livro (embora – e isto me deixa muito feliz – tenham sido muitos leitores) levaram-me a concluir que a estrutura do livro estava consolidada, não havendo, portanto, necessidade de ampliações naquela primeira edição, de sorte que para esta segunda edição foi feita apenas uma revisão quanto aos erros de ortografia e de gramática e incluídas algumas poucas observações e acrescentadas alguns poucos novos exercícios.

Basicamente, há duas correntes de pensamento em relação ao processo ensino/aprendizagem de programação de computadores. Uma delas defende o desenvolvimento de lógica de programação através de uma linguagem algorítmica, sem que haja necessidade de implementações em máquinas reais; a outra defende o desenvolvimento da lógica de programação concomitante com o estudo de uma linguagem de programação, o que permite a implementação de exercícios e de programas em computadores. A linguagem Pascal foi desenvolvida por Niklaus Wirth exatamente para os seguidores desta segunda corrente, tendo comandos com sintaxes simples e semânticas facilmente compreensíveis.

Atualmente, a linguagem Pascal, além de ser excelente para facilitar o desenvolvimento da lógica de programação, é a linguagem básica do Delphi, um dos ambientes visuais para desenvolvimento de sistemas de computação dos mais usados em todo o mundo.

Mantendo a estrutura da primeira, esta segunda edição é constituída de onze capítulos. O primeiro capítulo, *Introdução à Programação*, apresenta os conceitos básicos de computação necessários para aprendizagem de programação. O segundo capítulo, *Introdução à Linguagem Pascal*, apresenta os importantes conceitos de *variável* e de *tipo de dado*, as expressões aritméticas e lógicas e os comandos básicos da linguagem. Por sua vez, o capítulo três discute as *estruturas de decisão*, enquanto que o capítulo quatro discorre sobre as *estruturas de repetição*, estruturas fundamentais em programação e que propicia muitos exemplos que facilitam sobremaneira o desenvolvimento da lógica de programação. O estudo dos *procedimentos* e das *funções*, além do estudo da *recursividade* é o objetivo do quinto capítulo, enquanto que o capítulo seis estuda os *vetores* e as *matrizes* e o capítulo sete estuda as cadeias de caracteres. Já os capítulos oito, nove e dez estudam, respectivamente, os *registros* e *os arquivos*, *pesquisa* e *ordenação* e os *tipos de dados definidos pelo usuário* e os *conjuntos*. Para finalizar, o capítulo onze apresenta um estudo introdutório da *alocação dinâmica da memória*, base para a implementação de soluções de problemas mais complexos.

Além de apresentar com rigor as sintaxes dos comandos e situações práticas que motivam suas semânticas, o livro apresenta cerca de noventa e cinco exemplos, incluindo algoritmos, programas, funções e procedimentos e noventa e oito exercícios propostos, todos com respostas. Naturalmente, todos os programas, funções e procedimentos propostos devem ser implementados, não devendo o leitor, diante da primeira dificuldade, consultar a resposta. Pelo contrário, o aprendizando deve tentar de todas as formas encontrar suas soluções, podendo inclusive encontrar soluções melhores que aquelas sugeridas. Caso isto ocorra, e tenho certeza que ocorrerá muitas vezes, rogo o envio destas soluções para jaime@ccen.ufal.br para serem incluídas, com o devido crédito é claro, em futuras edições. Agradeceria também o recebimento de propostas de exercícios não contemplados aqui e qualquer sugestão ou crítica.

De acordo com o número de questões resolvidas que se discuta em sala de aula e com a

profundidade que se apresente alguns dos capítulos, este livro pode ser desenvolvido em cursos de sessenta a cento e vinte horas, cobrindo disciplinas iniciais de programação dos cursos da área de computação e informática (Ciência da Computação, Engenharia da Computação, Sistemas de Informações e Licenciatura em Informática) e dos cursos das ciências exatas (Engenharias, Matemática, Física, Meteorologia, etc.)

Por oportuno e por dever de gratidão, gostaria de agradecer a todos os estudantes do Curso de Ciência da Computação da Universidade Federal de Alagoas que, como alunos das minhas turmas de Programação 1 e de Tópicos de Matemática para Computação, tiveram participação fundamental no desenvolvimento dos meus livros ao me possibilitarem a aplicação das minhas idéias sobre programação, inclusive aperfeiçoando-as com sugestões sempre pertinentes. Gostaria também de registrar o apoio e o incentivo que sempre tenho recebido dos colegas professores dos Departamentos de Tecnologia da Informação e de Matemática da Universidade Federal de Alagoas, destacando, sem demérito para os demais, Ailton Cruz, Evandro Costa, Eliana Almeida, Sílvio Chagas, Washington Bonfim e Eduardo Perdigão. Registro também meu apreço e meus agradecimentos a Gorki Starlin que, desde o primeiro livro da série *Aprendendo a Programar* tem apoiado meu trabalho.

Por fim, gostaria de explicitar que todas as minhas ações são dedicadas ao meu lado feminino, representado por minha esposa Salete e minhas filhas Jaiane, Katiane e Aninha.

Em Maceió, no mês de maio do ano de 2004.

Jaime Evaristo

Sumário

Capítulo 1 Introdução à Programação

1.1 Organização básica de um computador

Um dos conceitos mais importantes para a programação de computadores é o conceito de *variável*, cuja compreensão requer um conhecimento básico da constituição de um computador.

Em linhas gerais, um computador é constituído de quatro unidades básicas: *unidade de entrada*, *unidade de saída*, *unidade de processamento central* e *memória*. Como indica sua denominação, uma *unidade de entrada* é um dispositivo que permite que o usuário interaja com o computador, fornecendo-lhe dados e informações. O *teclado* e o *mause* são os exemplos mais triviais de unidades de entrada. Uma *unidade de saída* serve para que sejam fornecidos ao usuário do computador os resultados do processamento realizado. O *monitor de vídeo* e uma *impressora* são exemplos de unidades de saída. A *unidade central de processamento* (*cpu*, acrossemia de ***c****entral* ***p****rocessing* ***u****nit*) é responsável por todo o processamento requerido. Nela são realizadas, por exemplo, operações aritméticas e lógicas.

A *memória* armazena dados e informações que serão utilizados no processamento, além dos *programas* que vão manipular estes dados e estas informações. Como veremos com um pouco mais de detalhes posteriormente, esta unidade é dividida em partes, chamadas *posições de memória*, sendo associada a cada uma delas um *endereço*, o qual é utilizado para se ter acesso à posição. Muitas vezes, esta unidade é chamada *memória RAM* (acrossemia de *random access memory*, memória de acesso aleatório) e quanto maior a sua capacidade de armazenamento, maior é a capacidade de processamento do computador. Qualquer armazenamento na memória de um computador é temporário, pois quando o computador é desligado tudo que está armazenado desaparece. Por esta razão se diz que se trata de uma memória *volátil*.

1.2 Linguagem de máquina

Como há fluxo de dados e informações entre as diversas unidades, há a necessidade de que elas se comuniquem. Por exemplo, um dado fornecido pelo teclado deve ser armazenado na memória; para a *cpu* realizar uma operação aritmética, ela vai buscar valores que estão armazenados na memória, e assim por diante. Para que haja comunicação entre as unidades do computador, é necessário que se estabeleça uma *linguagem de comunicação*. Os seres humanos, por exemplo, se comunicam basicamente através de duas linguagens: a linguagem escrita e a falada. Uma comunicação através de uma linguagem escrita é constituída de *parágrafos*, os quais contêm *períodos*, que contêm *frases,* que são constituídas de *palavras*, sendo cada uma das palavras formadas por *letras* e esta seqüência termina aí. Assim, uma *letra* é um ente indivisível da linguagem escrita e, em função disto, é chamada *símbolo básico* da linguagem escrita. Este exemplo foi apresentado para que se justifique a afirmação de que linguagens de comunicação, de um modo geral, requerem a existência de alguns símbolos básicos bem definidos. Os símbolos básicos da fala são os *fonemas* e, confesso, não sei se a *linguagem brasileira de sinais*, utilizada pelos deficientes auditivos, possui símbolos básicos.

Como a comunicação entre as unidades do computador teria que ser obtida através de fenômenos físicos, os cientistas que conceberam os computadores atuais estabeleceram dois símbolos básicos para a linguagem. Esta quantidade foi escolhida pelo fato de que, através de fenômenos físicos, é muito fácil obter dois estados distintos e não confundíveis, como passar corrente elétrica/não passar corrente elétrica, estar magnetizado/não estar magnetizado etc., podendo cada um destes estados ser um dos símbolos. Assim a linguagem utilizada para comunicação interna num computador, chamada *linguagem de máquina*, possui apenas dois símbolos. Cada um destes símbolos é denominado *bit* (de ***bi****nary digi****t***) e eles são representados por 0 (zero) e 1 (um). Esta forma de representar os *bit's* justifica a sua denominação: *binary digit* (dígito binário). Deste modo, as *palavras* da linguagem de máquina são seqüências de bits, ou seja, seqüências de dígitos zero e um. Isto explica a denominação *digital* para todo recurso que utilize esta linguagem: *computador digital*, *telefonia digital* etc.

Para que haja a possibilidade da comunicação do homem com o computador, é necessário que as palavras da linguagem escrita sejam traduzidas para a linguagem de máquina e vice-versa. Para que isto seja possível, é preciso que se estabeleça qual a seqüência de bit's que corresponde a cada caractere usado na

linguagem escrita. Ou seja, deve-se estabelecer uma codificação em seqüência de bit's para cada um dos caracteres. Uma codificação muito utilizada é o *código ASCII* (*American Standard Code for Information Interchange* ou *Código Padrão Americano para Intercâmbio de Informações*), estabelecido pelo *ANSI* (*American National Standard Institute*). Nesta codificação, cada caractere é representado por uma seqüência de oito bits (normalmente, um conjunto de oito bit's é chamado *byte*). Só para exemplificar, apresentamos a tabela abaixo com os códigos ASCII de alguns caracteres.

Tabela 1 Códigos ASCII de alguns caracteres

Caractere	Código ASCII
Espaço em branco	00100000
!	00100001
(	00100011
. . .	. . .
0	00110000
1	00110001
. . .	. . .
A	01000001
B	01000010
. . .	. . .
Z	01011010
. . .	. . .
a	01100001
. . .	. . .
z	01111010
. . .	. . .

Observe a necessidade de se haver codificado o *espaço em branco* (este "caractere" é utilizado para separar nossas palavras) e de se haver codificado diferentemente as letras maiúsculas e minúsculas para que se possa, caso se pretenda, considerá-las como coisas distintas.

Levando em conta que cada seqüência de zeros e uns pode ser vista como a representação de um número inteiro no *sistema binário de numeração* [Evaristo, J 2002], podemos, até para facilitar a sua manipulação, associar a cada código ASCII o inteiro correspondente, obtendo assim o que se costuma chamar de *código ASCII decimal*. Por exemplo, como 1000001 é a representação do número 65 no sistema binário de numeração, dizemos que o *código ASCII decimal* de A é 65. Uma pergunta que pode ser feita é por que o código ASCII da letra A foi escolhido 65 e não 1 ou 10 ou 100. A referência citada logo acima explica o porquê da escolha.

1.3 Algoritmos

Um conceito fundamental para a Ciência da Computação é o de *algoritmo*, cujo estudo formal é feito em disciplinas geralmente denominadas *Teoria da Computação*. Aqui veremos alguns aspectos informais desta idéia. Consideraremos um *algoritmo* como uma *seqüência de instruções*, cuja execução resulta na realização de uma determinada tarefa. De uma maneira natural, alguns tipos de algoritmos estão presentes no nosso dia-a-dia, não necessariamente envolvendo aspectos computacionais. Uma receita de bolo, uma partitura musical, um manual de utilização de um videocassete são algoritmos. As instruções de uma receita de bolo são do tipo "leve ao forno previamente aquecido", "bata as claras até ficarem em ponto de neve", etc. No caso de uma partitura musical, existem instruções que indicam que uma determinada nota musical deve ser executada por determinado tempo, enquanto que um manual de utilização de um videocassete contém instruções do tipo "selecione o canal desejado", "aperte a tecla *rec*", etc.

É claro que a execução de cada um destes algoritmos resulta na realização de alguma tarefa: a confecção de um bolo; a execução de uma melodia; a gravação de um programa de televisão. É claro também que a execução de cada um deles requer a utilização de alguns equipamentos constituídos de componentes elétricos, mecânicos e eletrônicos, como uma batedeira, um forno, um instrumento musical, uma televisão e um ser humano que seja capaz de interagir com os tais equipamentos para que estes executem as instruções. O conjunto de elementos que interagem para a execução de um algoritmo geralmente é chamado de *processador*, enquanto que um algoritmo em execução será chamado de *processo*. No exemplo da partitura musical, o processador é o conjunto {instrumentista, instrumento} e no caso de uma receita de cozinha o processador seria o conjunto constituído das panelas, do forno e da cozinheira. Naturalmente, o resultado do processo depende dos elementos que compõem o processador.

Evidentemente, o processador deve ser capaz de executar as instruções do algoritmo e o processo deverá parar em algum instante para que se tenha a realização da tarefa pretendida. Para que estas duas condições sejam satisfeitas, é necessário que um algoritmo satisfaça às seguintes exigências:

1.As instruções devem ser claras, não devendo conter ambigüidades, nem qualquer coisa que impeça sua execução pelo processador.

2.Não pode haver dubiedade em relação à próxima ação a ser realizada após a execução de uma determinada instrução.

3.Todas as instruções devem ser executadas num tempo finito.

Para exemplificar, considere o seguinte conjunto de instruções, que devem ser executadas seqüencialmente:

1.Sintonize, no videocassete, o canal desejado.
2.Insira uma fita no videocassete.
3.Acione a tecla *rec.*

À primeira vista, o conjunto de instruções acima constitui um algoritmo, visto que as exigências estabelecidas acima são todas satisfeitas. Veremos a seguir que podemos ter problemas na execução destas instruções. Antes, observe que a execução do suposto algoritmo requer a utilização de uma fita de videocassete. De forma semelhante, uma receita de bolo exige para sua execução os ingredientes. Isto significa que as execuções destes algoritmos necessitam de "dados" que serão fornecidos durante suas execuções. Estes dados constituem a *entrada* do algoritmo.

Naturalmente, e como foi dito acima, um algoritmo é desenvolvido para que, após a execução de suas instruções, uma determinada tarefa seja realizada. A realização desta tarefa é conhecida como a *saída* do algoritmo. A *saída* do algoritmo do exemplo anterior seria a gravação de um programa previamente escolhido; a *saída* de uma receita de bolo seria o bolo e a *saída* de da execução de uma partitura musical seria o som da melodia.

Introduzidos os conceitos de *entrada* e *saída* de um algoritmo, podemos pensar, de forma mais resumida, que um algoritmo é um conjunto de instruções que, recebendo uma *entrada* compatível com as instruções, fornece uma *saída* num tempo finito. A questão da entrada é que pode prejudicar o suposto algoritmo do exemplo acima. O que aconteceria se a fita que fosse inserida já tivesse sido utilizada e não houvesse sido previamente rebobinada? Evidentemente a gravação não seria realizada e a saída do algoritmo, para aquela entrada, não ocorreria. Seria necessária então uma instrução que posicionasse a fita no seu início independentemente da posição em que ela estivesse:

1.Sintonize, no videocassete, o canal desejado.
2.Insira uma fita no videocassete.
3.Acione a tecla *rr* {para rebobinar a fita}.
4.Acione a tecla *rec.*

É evidente que a execução da instrução 3 nem sempre resultará em alguma ação efetiva, o que pode ocorrer se a fita estiver na sua posição inicial. Este problema é resolvido precedendo a execução da instrução por um condicional:

3. Se a fita não estiver rebobinada, acione a tecla *rr*.

Naturalmente, teria de ser estabelecido algum procedimento que permitisse ao processador verificar se a fita estaria ou não rebobinada. Isto pode ser feito pelo ser humano que faz parte do processador, ou seja, pela

pessoa que está tentando gravar o programa. Assim, o algoritmo seria melhor se contivesse as seguintes instruções.

1. Sintonize no videocassete o canal desejado.
2. Apanhe uma fita, verifique se ela está rebobinada e a insira no videocassete.
3. Se a fita não estiver rebobinada, acione a tecla *rr*.
4. Acione a tecla *rec*

O algoritmo do exemplo acima (pelo menos, a maior parte dele) é definido pela própria construção do videocassete e nos é apresentado pelo manual de utilização do aparelho. Ou seja, o algoritmo está pronto e não há necessidade de que se pense. Não há necessidade de que se raciocine. Basta que se disponha dos equipamentos necessários e que se seja capaz de executar as instruções.

1.4 Lógica de programação

Estamos interessados em aprender a desenvolver algoritmos que resolvam problemas. E nestes casos é necessário pensar. Mais do que isto, é necessário raciocinar. É necessário aprender a raciocinar. E como aprender a raciocinar?

Embora para Copi, I. M. [Copi81] esta não seja uma definição completa, a ciência do raciocínio é a *lógica*. É esta ciência que estuda os princípios e métodos usados para distinguir os raciocínios corretos dos incorretos. Ao se estudarem estes métodos, aprende-se a raciocinar (inclusive diante de situações novas) e a, portanto, resolver problemas cuja solução não se conhecia. De um modo geral, a lógica estuda métodos que permitem estabelecer se um argumento de que uma certa afirmativa pode ser inferida a partir de outras afirmativas é ou não correto. Por exemplo, a lógica estuda métodos que garantem que a assertiva *João é racional* pode ser inferida a partir das afirmações *todo homem é racional* e *João é um homem*.

No dia-a-dia, usamos a palavra *lógica* para justificar um raciocínio que é indubitavelmente "razoável". É *lógico* que *se todo homem é um animal racional* e *João é um homem,* então *João é racional*. Ou seja, usamos a *lógica* para garantir a veracidade de uma afirmação a partir da veracidade de outras assertivas.

No nosso caso, queremos aprender a desenvolver algoritmos para resolver problemas. Queremos aprender a, dado um problema, determinar uma seqüência de instruções para um processador tal que, fornecidos os dados de entrada, a execução da seqüência de instruções redunde como saída a solução do problema. Embora isto não esteja consagrado ainda pela ciência, nem seja considerado como uma parte da lógica, o raciocínio que visa ao desenvolvimento de algoritmos é chamado *lógica de programação*. Por exemplo, imagine o seguinte problema: um senhor, infelizmente bastante gordo, está numa das margens de um rio com uma raposa, uma dúzia de galinhas e um saco de milho. O senhor pretende atravessar o rio com suas cargas, num barco que só comporta o senhor e uma das cargas. Evidentemente, o senhor não pode deixar em uma das margens, sozinhos, a raposa e a galinha, nem a galinha e o milho. A questão é escrever um algoritmo que oriente o senhor a realizar o seu intento. Naturalmente, na primeira viagem, ele não pode levar a raposa (neste caso, as galinhas comeriam o milho), nem o milho (caso em que a raposa devoraria as galinhas). Logo, na primeira viagem ele deve levar as galinhas. Como ele estará presente na chegada, na segunda viagem ele pode levar a raposa ou o milho. Mas, e a volta para apanhar terceira carga? A solução é ele voltar com as galinhas! E aí, atravessar o milho, já que não há problema em que a raposa e o milho fiquem juntos. Escrevendo as instruções na seqüência em que elas devem ser executadas, teremos o seguinte algoritmo.

1. Atravesse as galinhas.
2. Retorne sozinho.
3. Atravesse a raposa.
4. Retorne com as galinhas.
5. Atravesse o milho.
6. Retorne sozinho.
7. Atravesse as galinhas.

1.5 Resolução de problemas

Uma pergunta que o leitor pode estar se fazendo é: como vou "descobrir" que a primeira instrução deve

ser a travessia das galinhas?

Algumas tarefas para as quais se pretende escrever um algoritmo podem ser vistas como um problema a ser resolvido. O exemplo anterior é um exemplo claro de uma tarefa com esta característica. Existem algumas técnicas que podem ser utilizadas para a resolução de problemas. No exemplo anterior, para se definir qual seria a primeira instrução, como existem apenas três possibilidades, verifica-se o que aconteceria ao se escolher determinada instrução. Foi o que, de passagem, foi feito acima: se o homem atravessasse primeiro o milho, a raposa devoraria as galinhas; se o homem atravessasse primeiro a raposa, as galinhas comeriam o milho. Neste caso, podemos dizer que foi utilizada a técnica da exaustão: como o número de alternativas era pequeno, analisamos todas elas, uma a uma.

Esta técnica pode ser utilizada também na solução do seguinte problema: dispõe-se de três esferas idênticas na forma, sendo duas delas de mesmo peso e a terceira de peso maior. A questão é descobrir qual a esfera de peso diferente, realizando-se apenas uma pesagem numa balança de dois pratos. Para isto chamemos de A e B as esferas de mesmo peso e de C a de maior peso. Se optarmos por colocar duas esferas num dos pratos e a outra esfera no outro, temos as seguintes possibilidades:

a) (A+B, C).
b) (A+C, B).
c) (B+C, A).

No primeiro caso, pode acontecer qualquer coisa: a balança pode ficar equilibrada, se Peso(C) = Peso(A+B); ficar inclinada para o lado esquerdo, se Peso(C) > Peso(A+B) ou ficar inclinada para o lado direito se Peso(C) < Peso(A+B). Observe que nada pode distinguir a esfera C. Nos dois últimos casos, a balança se inclinará para a esquerda, mas, outra vez, nada distingue a esfera C. Por exaustão, resta então escolhermos duas esferas e colocarmos cada uma delas num dos pratos da balança. Temos então as seguintes possibilidades:

a) (A, B).
b) (A, C).
c) (B, C).

No primeiro caso, a balança ficará equilibrada, o que indica que a mais pesada é aquela não escolhida; nos outros dois casos, a balança se inclinará para a direita, indicando que a esfera mais pesada é aquela que ocupa o prato respectivo. Temos então o seguinte algoritmo:

1. Escolha duas esferas.
2. Coloque cada uma das esferas escolhidas num dos pratos da balança.
3. Se a balança ficar equilibrada, forneça como resposta a esfera não escolhida; caso contrário, forneça como resposta a esfera do prato que está num nível mais baixo.

Uma outra técnica de resolução de problemas consiste em se tentar resolver casos particulares da questão ou resolver a questão para dados menores do que os dados que foram fixados. Para exemplificar, consideremos a seguinte questão: como obter exatamente 4 litros de água dispondo de dois recipientes com capacidades de 3 litros e 5 litros[1]? Como 4 = 3 + 1 ou 4 = 5 – 1 conseguiremos resolver a questão se conseguirmos obter 1 litro. Mas isto é fácil, pois 1 = 3 + 3 – 5! Temos então o seguinte algoritmo:

1. Encha o recipiente de 3 litros.
2. Transfira o conteúdo do recipiente de 3 litros para o recipiente de 5 litros.
3. Encha o recipiente de 3 litros.
4. Com o conteúdo do recipiente de 3 litros, complete o recipiente de 5 litros.
5. Esvazie o recipiente de 5 litros.
6. Transfira o conteúdo do recipiente de três litros para o recipiente de 5 litros.
7. Encha o recipiente de 3 litros.
8. Transfira o conteúdo do recipiente de 3 litros para o recipiente de 5 litros.

Para compreender o algoritmo, sejam A e B os recipientes de 3 litros e de 5 litros, respectivamente, e indiquemos por (X, n) o fato de o recipiente *X* conter *n* litros de água. No início temos (A, 0) e (B, 0) e, após a execução de cada instrução, teremos:

1. (A, 3), (B, 0).

[1] A solução desta questão foi necessária no filme Duro de Matar 3 para um policial desativar uma bomba.

2. (A, 0), (B, 3).
3. (A, 3), (B, 3).
4. (A, 1), (B, 5).
5. (A, 1), (B, 0).
6. (A, 0), (B, 1).
7. (A, 3), (B, 1).
8. (A, 0), (B, 4).

Outras questões que podem ser levantadas são: há outras soluções? Existe alguma solução que realize a mesma tarefa com menos instrução? Para responder a estas questões talvez seja interessante lembrar que 4 = 5 – 1. Significa que, se conseguirmos tirar 1 litro do recipiente de 5 litros quando ele estiver cheio, resolveremos a questão. Para conseguir isto, basta que o recipiente de 3 litros contenha 2 litros. E para se obter 2 litros? Aí basta ver que 2 = 5 – 3. Podemos então resolver a questão com o seguinte algoritmo:

1. Encha o recipiente de 5 litros.
2. Com o conteúdo do recipiente de 5 litros, encha o de 3 litros.
3. Esvazie o recipiente de 3 litros.
4. Transfira o conteúdo do recipiente de 5 litros para o recipiente de 3 litros.
5. Encha o recipiente de 5 litros.
6. Com o conteúdo do recipiente de 5 litros, complete o recipiente de 3 litros.

Após a execução de cada uma das instruções teremos:

1. (A, 0), (B, 5).
2. (A, 3), (B, 2).
3. (A, 0), (B, 2).
4. (A, 2), (B, 0).
5. (A, 2), (B, 5).
6. (A, 3), (B, 4).

Uma outra técnica bastante utilizada é se tentar raciocinar a partir de uma solução conhecida de uma outra questão. Para compreender isto considere as duas seguintes questões: imagine uma relação de *n* números, os quais podem ser referenciados por a_i com i = 1, 2, ..., n e queiramos somá-los com a restrição de que só sabemos efetuar somas de duas parcelas. Para resolver esta questão, podemos pensar em casos particulares: se n = 2, basta somar os dois números; se n = 3, basta somar os dois primeiros e somar esta soma com o terceiro. Naturalmente este raciocínio pode ser reproduzido para n > 3. A questão é que a soma dos dois primeiros deve estar "guardada" para que se possa somá-la com o terceiro, obtendo-se a soma dos três primeiros; esta soma deve ser "guardada" para que seja somada com o quarto e assim sucessivamente. Para isto podemos estabelecer uma *referência* à soma "atual", a qual será alterada quando a soma com o elemento seguinte for efetuada. Até para somar os dois primeiros, pode-se pensar em somar "a soma do primeiro" com o segundo.

Temos então o seguinte algoritmo:

1. Faça i = 1.
2. Faça Soma = a_1.
3. Repita n – 1 vezes as instruções 3.1 e 3.2.
    3.1. Substitua i por i + 1.
    3.2. Substitua Soma por Soma + a_i.

Por exemplo: se n = 5 e a_1 = 8, a_2 = 4, a_3 = 9, a_4 = 13 e a_5 = 7, a execução do algoritmo resultaria nas seguintes ações:

1. i = 1.
2. Soma = 8.
3.1.1. i = 2.
3.2.1. Soma = 8 + 4 = 12
3.1.2. i = 3.
3.2.2. Soma = 12 + 9 = 21.
3.1.3. i = 4.
3.2.3. Soma = 21 + 13 = 34.
3.1.4. i = 5.

3.2.4. Soma = 34 + 7 = 41.

Como veremos ao longo do livro, este algoritmo é bastante utilizado em programação, sendo mais comum o primeiro termo da relação ser "somado" dentro da repetição. Neste caso, para que o primeiro seja somado, é necessário que *Soma* seja inicializado com 0 (zero), ficando assim o algoritmo:

1. Faça i = 0.
2. Faça Soma = 0.
3. Repita n vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Soma por Soma + a_i.

Conhecendo este algoritmo, é fácil então resolver a questão de se calcular o produto de *n* números nas condições, e aí vemos como utilizar uma solução conhecida para resolver um problema. Deve-se inicializar uma referência *Produto* com 1 e, numa repetição, multiplicar os números como foi feito no caso da soma:

1. Faça i = 0.
2. Faça Produto = 1.
3. Repita n vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Produto por Produto x a_i.

1.6 Exemplos de algoritmos

Alguns dos problemas anteriores são importantes para se desenvolver o raciocínio, mas não é este tipo de problema que se pretende discutir aqui. Estamos interessados em algoritmos para:

1. Resolver problemas matemáticos, como um algoritmo para determinar a média aritmética de vários números dados; determinar as raízes de uma equação do segundo grau; encontrar o máximo divisor comum de dois números dados.

2. Resolver questões genéricas, como um algoritmo para colocar em ordem alfabética uma relação de nomes de pessoas; atualizar o saldo de uma conta bancária na qual se fez um depósito; corrigir provas de um teste de múltipla escolha; um algoritmo para se cadastrar um novo usuário de uma locadora, etc..

O algoritmo para o cálculo da média pode ser escrito de forma muito simples:

1. Determine a quantidade de números;
2. Some os números dados;
3. Divida esta soma pela quantidade de números.

Obviamente, a entrada deste algoritmo será uma relação de números e a saída será a média aritmética destes números. Naturalmente qualquer pessoa que saiba contar, somar e dividir números é capaz de executar este algoritmo dispondo apenas de lápis e papel. A questão que se põe é: e se a relação contiver 13 426 números? A tal pessoa é capaz de executar, porém, quanto tempo levará para fazê-lo?

Um outro aspecto a ser observado é que nem sempre a linguagem coloquial é eficiente para se escreverem as instruções. Nessa linguagem o algoritmo para determinação das raízes de uma equação do segundo grau teria uma instrução difícil de escrever e difícil de compreender como:

n. Subtraia do quadrado do segundo coeficiente o produto do número quatro pelo produto dos dois outros coeficientes.

Isto pode ser parcialmente resolvido utilizando-se uma linguagem próxima da linguagem matemática, considerando a entrada e valores intermediários como se constituíssem um conjunto de variáveis. No caso da equação do segundo grau, como a entrada seria o conjunto dos valores dos coeficientes, poderíamos ter o seguinte algoritmo, que nos foi apresentado nas séries finais do nosso ensino fundamental:

1. Chame de a, b e c os coeficientes da equação.
2. Calcule d = b² - 4ac.
3. Se d < 0 forneça como resposta a mensagem: A equação não possui raízes reais.
4. Se d · 0
   4.1 Calcule x' = (-b + raiz(d))/2a e x' = (-b - raiz(d))/2ª.

4.2 Forneça x' e x' como raízes da equação.

De maneira mais ou menos evidente, *raiz(d)* está representando a raiz quadrada de *d* e a execução deste algoritmo requer que o processador seja capaz de determinar valores de expressões aritméticas, calcular raízes quadradas, efetuar comparações e que conheça a linguagem matemática.

É interessante notar que o algoritmo para o cálculo da média aritmética, que é tão simples na linguagem coloquial, não o é na "linguagem matemática". A dificuldade resulta do fato de que, em princípio, não se conhece a quantidade de números, o que impediria de se considerar a entrada como um conjunto de variáveis. Logo mais veremos como solucionar esta questão.

Algoritmos para problemas genéricos são mais complicados e a linguagem utilizada acima não é suficiente (para o caso da ordenação de uma relação de nomes, foram desenvolvidos vários algoritmos e teremos oportunidade de discutir alguns deles ao longo deste livro).

Os exemplos discutidos acima mostram que a elaboração de um algoritmo exige que se conheça o conjunto de instruções que o processador é capaz de executar. Alguns autores de livros com objetivos idênticos a este - facilitar a aprendizagem da programação de computadores - iniciam seus textos discorrendo exclusivamente sobre *resolução de problemas*, encarando o processador como uma "caixa preta" que recebe as instruções formuladas pelo algoritmo e fornece a solução do problema, não levando em conta o processador quando da formulação do tal algoritmo. Entendemos que esta não é a melhor abordagem, visto que o conhecimento do conjunto de instruções que o processador pode executar pode ser definidor na elaboração do algoritmo. Por exemplo: imagine que queiramos elaborar um algoritmo para extrair o algarismo da casa das unidades de um inteiro dado (apresentaremos posteriormente uma questão bastante prática cuja solução depende deste algoritmo). Evidentemente, o algoritmo para resolver esta "grande" questão depende do processador que vai executá-lo. Se o processador for um ser humano que saiba o que é número inteiro, algarismo e casa das unidades, o algoritmo teria uma única instrução:

1. Forneça o algarismo das unidades do inteiro dado.

Porém, se o processador for um ser humano que saiba o que é número inteiro e algarismo, mas não saiba o que é casa das unidades, o algoritmo não poderia ser mais esse. Neste caso, para resolver a questão, o processador deveria conhecer mais alguma coisa, como, por exemplo, ter a noção de "mais à direita", ficando o algoritmo agora como:

1. Forneça o algarismo "mais à direita" do número dado.

E se o processador não sabe o que é algarismo, casa das unidades, "mais à direita", etc.? Nesta hipótese, quem está elaborando o algoritmo deveria conhecer que instruções o processador é capaz de executar para poder escrever o seu algoritmo. Por exemplo, se o processador é capaz de determinar o resto de uma divisão inteira, o algoritmo poderia ser:

1. Chame de n o inteiro dado;
2. Calcule o resto da divisão de n por 10;
3. Forneça este resto como o algarismo pedido.

E se o sistema não fosse capaz de calcular o resto de uma divisão inteira? A resposta a esta pergunta está embutida na solução da questão abaixo, que discutiremos como mais um exemplo. Nesta discussão (e na discussão dos demais algoritmos), vamos supor que o processador é capaz de:

1. Representar por seqüências de caracteres (chamadas *variáveis*) valores numéricos (que passam a ser chamados *valores* da variável).

2. Resolver expressões aritméticas que envolvam as operações de multiplicação, divisão, soma e subtração.

3. Realizar comparações entre dois valores.

4. Repetir a execução de um conjunto de instruções uma quantidade fixada de vezes ou até que uma condição seja atingida.

5. Atribuir um valor a uma variável.

6. Substituir o valor de uma variável por outro valor.

7. Fornecer o valor de uma variável ou emitir uma mensagem.

A questão a ser discutida é a determinação do quociente e do resto da divisão de dois inteiros positivos dados. Por exemplo: se a entrada for 30 para o dividendo e 7 para o divisor, a saída deve ser 4 para o quociente e 2 para o resto. Fomos ensinados que, para determinar o quociente, deveríamos, por tentativa, encontrar o número que multiplicado pelo divisor resultasse no maior número menor que o dividendo. No exemplo numérico citado, poderíamos tentar o 5 e teríamos 5x7 = 35 que é maior que 30; tentaríamos o 3 obtendo 3x7 = 21 que talvez seja pequeno demais em relação ao 30; aí tentaríamos o 4 obtendo 4x7 = 28, encontrando então o quociente 4.

Um algoritmo para solucionar esta questão poderia ser:

1. Chame de D1 e D2 o dividendo e o divisor dados.
2. Faça I = 1.
3. repita 3.1 até IxD2 > D1.
    3.1. Substitua I por I + 1.
4. Calcule Q = I – 1.
5. Calcule r = D1 - QxD2.
6. Forneça r para o resto e Q para o quociente pedidos.

No exemplo numérico proposto, teríamos a seguinte tabela com os valores obtidos durante a execução do algoritmo:

D1	D2	I	QxI	Q	r
30	7				
		1	7		
		2	14		
		3	21		
		4	28		
		5	35		
				4	2

Observando os exemplos acima discutidos, pode-se resumir algumas características comuns a algoritmos:

1. O processador deve ser capaz de executar as instruções. Para isto elas devem ser escritas, de forma clara e precisa, numa linguagem compreendida pelo processador.

2. As instruções são executadas seqüencialmente.

3. Algoritmos podem manipular valores que serão fornecidos na hora de sua execução. Estes valores são chamados *dados de entrada*, sendo o conjunto destes dados denominado *entrada* do algoritmo.

4. Algumas instruções podem manipular valores gerados pelo próprio algoritmo em instruções anteriores.

5. A execução de algumas instruções depende da análise de alguma condição.

6. Algumas instruções devem ter suas execuções repetidas.

7. É necessária a existência de instruções que forneçam os resultados da execução do algoritmo. Estes resultados constituem a *saída* do algoritmo.

1.7 Mais exemplos de algoritmos

1. Como se depreende facilmente da sua denominação, o *máximo divisor comum* (*mdc*) de dois números dados é o maior número que os divide. Antes o *mdc* só era utilizado para simplificações de frações ordinárias; atualmente ele é utilizado na determinação de *chaves públicas* para sistemas de criptografia RSA [Evaristo, J, 2002]. Por exemplo, mdc(64, 56) = 8. De maneira óbvia, o algoritmo abaixo determina o *mdc* de dois números dados:

1. Chame de x e de y os números.
2. Determine D(x), o conjunto dos divisores de x.
3. Determine D(y), o conjunto dos divisores de y.
4. Determine I, a interseção de D(x) e D(y).

5. Determine M, o maior elemento do conjunto I.
6. Forneça M como o mdc procurado.

O cálculo de mdc(64, 56) com este algoritmo seria:

1. x = 64, y = 56.
2. D(64) = {1, 2, 4, 8, 16, 32, 64}.
3. D(56) = {1, 2, 4, 7, 8}.
4. I = {1, 2, 4, 8}.
5. M = 8.

O interessante é que mostraremos posteriormente que este algoritmo, bastante fácil na sua compreensão, é computacionalmente bastante ineficiente no sentido de que sua execução pode, dependendo dos valores de x e y, demandar um tempo acima do razoável.

Por incrível que possa parecer, o algoritmo mais eficiente para o cálculo do máximo divisor comum de dois números foi desenvolvido pelo matemático grego Euclides duzentos anos antes de Cristo. O *algoritmo de Euclides* nos é apresentado nas séries intermediárias do ensino fundamental através de um esquema como o diagrama do exemplo abaixo, cujo objetivo é encontrar o máximo divisor comum de 204 e 84.

	2	2	3
204	84	36	12
36	12	0	

O esquema funciona da seguinte forma: divide-se 204 por 84 obtendo-se resto 36; a partir daí, repetem-se divisões até que o resto seja zero, sendo o dividendo da divisão atual o divisor da divisão anterior e o divisor da divisão atual o resto da divisão anterior. O último divisor é o máximo divisor procurado. Como se pode ver, estas instruções escritas desta forma não são nada compreensíveis, o que faz com elas sejam transmitidas oralmente nas salas do ensino fundamental. No capítulo 4 (quatro), teremos a oportunidade de discutir este algoritmo com detalhes e veremos que ele é um algoritmo bastante interessante no desenvolvimento da lógica de programação e que, às vezes, um algoritmo é mais compreensível que a linguagem coloquial.

2. Discutiremos agora o algoritmo para o cálculo da média de uma relação contendo um número grande (digamos, 10 000) de números dados. No caso da equação do segundo grau, eram três os dados de entrada e, portanto, os chamamos de *a*, *b*, e *c*. Mas agora são 10 000 os dados de entrada! Uma solução possível é receber os números um a um, somando-os antes de receber o seguinte, conforme vimos na seção 1.5.

1. Chame de A o primeiro número dado.
2. Faça S = A.
3. Repita 9 999 vezes as instruções 3.1 e 3.2.
   3.1 Chame de A o próximo número dado.
   3.2 Substitua o valor de S por S + A.
4. Calcule M = S/10 000.
5. Forneça M para o valor da média.

Por exemplo, se a relação de números fosse {5, 3, 8, 11, ...} até a quarta execução de 3.1 e 3.2 teríamos a seguinte tabela:

A	S	M
5	5	
3	8	
8	16	
11	27	

Está fácil perceber que após 9 999ª execução das instruções 3.1 e 3.2 a variável S conterá a soma de todos os números da relação, o que justifica a instrução 4.

3. Um outro exemplo que justifica plenamente a necessidade do conhecimento do que o processador é capaz de executar é a determinação do maior número de uma relação de números. Se o processador for um aluno do ensino médio e a relação contiver poucos números, uma simples olhada na relação permitirá se identificar o maior número. Mas, e se o processador for um aluno das classes iniciais do ensino fundamental? E se a relação contiver 10 000 números? E se os números estiverem escritos em forma de fração ordinária?

Uma solução possível é supor que o maior número é o primeiro da relação e comparar este suposto maior

com os demais números, alterando-o quando for encontrado um número na relação maior do que aquele que até aquele momento era o maior.

1. Chame de A o primeiro número dado.
2. Faça M = A.
3. Repita 9 999 vezes as instruções 3.1 e 3.2.
   3.1 Chame de A o próximo número dado.
   3.2 Se A > M substitua o valor de M por A.
4. Forneça M para o valor do maior número.

Para exemplificar, suponha que a entrada fosse o conjunto {5, 3, 8, 11, 10...}. Até a quinta execução das instruções 3.1 e 3.2 teríamos a seguinte tabela:

A	M
5	5
3	
8	8
11	11
10	

1.8 Linguagens de alto nível

Computadores digitais foram concebidos para executar instruções escritas em linguagem de máquina. Ou seja, um computador é capaz de executar um algoritmo contendo instruções escrita como seqüências de bits. Nos primórdios da computação, os algoritmos que se pretendiam que fossem executados por um computador eram escritos em linguagem de máquina, o que tornava a tarefa de desenvolvimento de algoritmos muito trabalhosa. A dificuldade vinha do fato de que era necessário que se conhecesse qual seqüência de bits correspondia à instrução pretendida, e este conhecimento era dificultado pelo fato de que o ser humano não está habituado com uma linguagem com apenas dois símbolos básicos.

Um grande avanço ocorreu na computação quando se conseguiu criar programas que traduzissem instruções escritas originariamente numa linguagem dos seres humanos para a linguagem de máquina. O surgimento de programas para esta finalidade permitiu o desenvolvimento de algoritmos em linguagens que utilizam caracteres, palavras e expressões de um idioma, ou seja, uma linguagem cujos símbolos básicos e cujas palavras estão no nosso cotidiano. Uma linguagem com esta característica é chamada *linguagem de alto nível*, sendo que *alto nível* aí não se refere à qualidade e sim ao fato de que ela está mais próxima da linguagem do ser humano do que da linguagem da máquina (quando alguma coisa está mais próxima da máquina do que do ser humano, dizemos que ela é *de baixo nível*). Como exemplo de linguagens de alto nível, temos *Pascal*, *C*, *Delphi*, *Visual Basic, Java* e C^{++}. Um algoritmo escrito numa linguagem de alto nível é chamado *programa fonte* ou simplesmente *programa*.

Como foi dito acima, um *programa fonte* deve ser traduzido para a linguagem de máquina. Há dois tipos de programas que fazem isto: os *interpretadores* que traduzem as instruções para a linguagem de máquina uma a uma e os *compiladores* que traduzem todo o programa. Um compilador ao receber como entrada um programa fonte, fornece como saída um programa escrito em linguagem de máquina chamado *programa objeto*. Assim a *compilação* de programa fonte gera um programa que pode então ser executado pelo computador. É comum nos referirmos à execução do programa fonte quando se está executando na realidade o programa objeto.

Como um interpretador traduz para a linguagem de máquina as instruções do programa fonte uma a uma, executando-as em seguida, a *interpretação* de um programa não gera um programa objeto.

1.9 Sintaxe e semântica de uma instrução

Dissemos que um programa escrito em linguagem de alto nível é traduzido para a linguagem de máquina por um compilador ou cada instrução é traduzida por um interpretador. É natural se admitir que, para que o compilador consiga traduzir uma instrução escrita com caracteres de algum idioma para instruções escritas como seqüências de zeros e uns, é necessário que cada instrução seja escrita de acordo com regras preestabelecidas. O conjunto destas regras é chamado *sintaxe* da instrução e quando não são obedecidas dizemos que existe *erro de sintaxe*.

Se o programa fonte contém algum erro de sintaxe, o compilador não o traduz para a linguagem de máquina (isto é, o compilador não *compila* o programa) e indica qual o tipo de erro cometido e a instrução onde este erro aconteceu. Se o programa fonte for interpretado, ele é executado até a instrução que contém erro de sintaxe quando então é interrompida a sua execução e o erro é indicado.

Naturalmente, cada instrução tem uma finalidade específica: a execução de uma instrução resulta na realização de alguma ação, digamos *parcial*, sendo a seqüência das ações parciais que redunda na realização da tarefa para a qual o programa foi escrito. A ação resultante da execução de uma instrução é chamada *semântica* da instrução. Um programa pode não conter erros de sintaxe (e, portanto, pode ser executado), mas a sua execução pode não fornecer como saída o resultado esperado para alguma entrada. Neste caso, dizemos que o programa contém *erros de lógica* que, ao contrário dos erros de sintaxe que são detectados pelo compilador ou pelo interpretador, são, às vezes, de difícil detecção.

No nosso entendimento, para se aprender a programar numa determinada linguagem é necessário que se aprendam as instruções daquela linguagem (para que se conheça o que o processador é capaz de fazer), a sintaxe de cada uma destas instruções e as suas semânticas. Aliado a isto se deve ter um bom desenvolvimento de *lógica programação* para que se escolham as instruções necessárias e a seqüência segundo a qual estas instruções devem ser escritas, para que o programa, ao ser executado, execute a tarefa pretendida. Felizmente ou infelizmente, para cada tarefa que se pretende não existe apenas uma seqüência de instruções que a realize. Ou seja, dado um problema, não existe apenas um programa que o resolva. Devemos procurar o *melhor programa*, entendendo-se como *melhor programa* um programa que tenha boa *legibilidade*, cuja execução demande o menor tempo possível e que necessite, para sua execução, a utilização mínima da memória.

Existe um conjunto de instruções que é comum a todas as linguagens de alto nível e cujas semânticas permitem executar a maioria das tarefas. A aprendizagem das semânticas destas instruções e das suas sintaxes em alguma linguagem de programação (aliada ao desenvolvimento da *lógica de programação*, desculpem-me a repetição) permite que se aprenda com facilidade qualquer outra linguagem do mesmo paradigma.

1.10 Sistemas de computação

Como foi dito anteriormente, a *cpu* de um computador é capaz de executar instruções (escritas em linguagem de máquina, permitam a repetição), ou seja, um computador é capaz de executar programas e só para isto é que ele serve. Se um computador não estiver executando um programa, ele para nada está servindo. Como foram concebidos os computadores atuais, um programa para ser executado deve estar armazenado na sua *memória*. O armazenamento dos programas (e todo o gerenciamento das interações entre as diversas unidades do computador) é feito por um programa chamado *sistema operacional*. Um dos primeiros sistemas operacionais para gerenciamento de microcomputadores foi o *DOS(Disk Operating System)*. Quando um computador é ligado, de imediato o sistema operacional é armazenado na memória e só a partir daí o computador está apto a executar outros programas. Estes programas podem ser:

- um *game*, que transforma o "computador" num poderoso veículo de entretenimento;
- um *processador de texto*, que transforma o "computador" num poderoso veículo de edição de textos;
- uma *planilha eletrônica*, que transforma o "computador" num poderoso veículo para manipulação de tabelas numéricas;
- programas para gerenciar, por exemplo, o dia-a-dia comercial de uma farmácia;
- ambientes que permitam o desenvolvimento de *games* ou de programas para gerenciar o dia-a-dia comercial de uma farmácia;

Talvez com exceção de um *game*, os programas citados acima são, na verdade, conjuntos de programas que podem ser executados de forma integrada. Um conjunto de programas que podem ser executados de forma integrada é chamado *software*. Por seu turno, as unidades do computador, associadas a outros equipamentos chamados *periféricos*, como uma impressora, constituem o *hardware*. O que nos é útil é um conjunto *software* + *hardware*. Um conjunto deste tipo é chamado de um *sistema de computação*. De agora

em diante, os nossos processadores serão *sistemas de computação*: queremos escrever programas que sejam executados por um *sistema de computação*.

Como foi dito acima, o desenvolvimento de um programa que gerencie o dia-a-dia comercial de uma farmácia requer um compilador (ou um interpretador) que o traduza para a linguagem de máquina. Antigamente as empresas que desenvolviam compiladores desenvolviam apenas estes programas, de tal sorte que o programador necessitava utilizar um processador de texto à parte para edição do programa fonte. Atualmente os compiladores são integrados num sistema de computação que contém, entre outros:

1. *Processador de texto*, para a digitação dos programas fontes;

2. *Depurador*, que permite que o programa seja executado instrução a instrução, o que facilita a descoberta de erros de lógica;

3. *Help*, que descreve as sintaxes e as semânticas de todas as instruções da linguagem e que descreve outros recursos do sistema;

4. *Linker*, que permite que um programa utilize outros programas.

Rigorosamente falando, um sistema constituído de um compilador e os *softwares* listados acima deveria ser chamado de *ambiente de programação*; é mais comum, entretanto, chamá-lo simplesmente de *compilador*.

O ambiente de programação que utilizamos para desenvolver os programas deste livro foi o compilador Turbo Pascal, versão 7.0, desenvolvido pela *Borland International, Inc.*, em 1992. Como se pode ver, é um sistema desenvolvido há bastante tempo (as coisas em computação andam muito mais rápido), já estando disponível gratuitamente na *internet.*

1.11 Por que a linguagem Pascal?

Há duas correntes de pensamento em relação à aprendizagem de programação de computadores e do desenvolvimento da lógica de programação. Uma delas advoga que esta aprendizagem seja desenvolvida numa linguagem algorítmica cujas instruções sejam escritas em português, cujas sintaxes indiquem facilmente suas semânticas e que sejam escritas de forma estruturada, no sentido de que, excetuando as *estruturas de repetição*, as instruções sejam executadas seqüencialmente (usualmente denominamos uma linguagem algorítmica com estes requisitos de um *português estruturado*). Para esta corrente, deve-se desenvolver a lógica de programação num português estruturado e, em seguida, realizar-se o estudo de uma linguagem de programação específica, enfatizando mais as sintaxes das instruções desta linguagem já que a lógica de programação já havia sido desenvolvida. Para os seguidores desta corrente, Evaristo, J. e Crespo, S. escreveram o livro *Aprendendo a Programar Programando numa Linguagem Algorítmica Executável (ILA)*, que desenvolve a lógica de programação, utilizando um português estruturado e um interpretador desta linguagem algorítmica (ILA) [Evaristo, J, Crespo, S. 2000].

A outra corrente defende que o desenvolvimento da lógica de programação seja efetuado diretamente numa linguagem de programação de objetivos gerais. Esta defesa baseia-se no fato de que uma linguagem como esta oferece diversos recursos de programação, o que facilita alcançar o objetivo da aprendizagem.

Este livro se destina aos adeptos desta segunda corrente e utiliza a linguagem Pascal pelo fato de ela ter sintaxes simples e intuitivas, tendo sido concebida por Niklaus Wirth, com o propósito inicial de facilitar o ensino de programação, objetivo este alcançado plenamente. Além disso, o aprimoramento que as empresas desenvolvedoras de compiladores lhe propiciaram permitiu que ela atingisse um patamar de linguagem de programação para todos os objetivos, podendo hoje ser considerada como uma das principais linguagens de programação existentes. Essa linguagem é também a linguagem básica da linguagem *Object Pascal* (*Pascal orientado a objetos*) utilizada no *Borland Delphi*, um ambiente de programação visual dos mais utilizados no mundo. Ainda mais, a aprendizagem de Pascal pode ser realizada num ambiente mais amigável, pois os programas aqui discutidos podem ser desenvolvidos no *Pascal for Windows*, uma ferramenta cuja interface com o usuário é mais interessante que a interface do Turbo Pascal 7.0.

1.12 Exercícios propostos

1. Três índios, conduzindo três brancos, precisam atravessar um rio dispondo para tal de um barco cuja

capacidade é de apenas duas pessoas. Por questões de segurança, os índios não querem ficar em minoria, em nenhum momento e em nenhuma das margens. Escreva um algoritmo que oriente os índios para realizarem a travessia nas condições fixadas. (Cabe observar que, usualmente, este exercício é enunciado envolvendo três jesuítas e três canibais. A alteração feita é uma modesta contribuição pessoal para o resgate da verdadeira história dos índios).

2. O jogo conhecido como *Torre de Hanói* consiste de três torres chamadas *origem*, *destino* e *auxiliar* e um conjunto de *n* discos de diâmetros diferentes, colocados na torre *origem* na ordem decrescente dos seus diâmetros. O objetivo do jogo é, movendo um único disco de cada vez e não podendo colocar um disco sobre outro de diâmetro menor, transportar todos os discos para torre *destino*, podendo usar a torre *auxiliar* como passagem intermediária dos discos. Escreva algoritmos para este jogo nos casos n = 2 e n = 3.

3. Imagine que se disponha de três esferas numeradas 1, 2 e 3 iguais na forma, duas delas com pesos iguais e diferentes do peso da outra. Escreva um algoritmo que, com duas pesagens numa balança de dois pratos, determine a esfera de peso diferente e a relação entre seu peso e o peso das esferas de pesos iguais.

4. A *média geométrica* de *n* números positivos é a raiz n-ésima do produto destes números. Supondo que o processador é capaz de calcular raízes n-ésimas, escreva um algoritmo para determinar a média geométrica de *n* números dados.

5. Sabendo que o dia 01/01/1900 foi uma segunda-feira, escreva um algoritmo que determine o dia da semana correspondente a uma data, posterior a 01/01/1900, dada. Por exemplo, se a data dada for 23/01/1900, o algoritmo deve fornecer como resposta terça-feira.

6. O show de uma banda de rock, que será realizado na margem de um rio, deve começar exatamente às 21 h. Atrasados, às 20 h 43, os quatro integrantes da banda estão na outra margem do rio e necessitam, para chegar ao palco, atravessar uma ponte. Há somente uma lanterna e só podem passar uma ou duas pessoas juntas pela ponte, e sempre com a lanterna. A lanterna não pode ser jogada e cada integrante possui um tempo diferente para atravessar a ponte: o vocal leva 10 minutos, o guitarrista 5 minutos, o baixista 2 minutos e o baterista 1 minuto. Evidentemente, quando dois atravessam juntos, o tempo necessário é o do mais lento. Escreva um algoritmo que permita que a banda atravesse a ponte de modo que o show comece na hora marcada.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 2 Introdução à Linguagem Pascal

2.1 Variáveis simples

Na seção 1.6 admitimos que o nosso processador era capaz de associar cadeias de caracteres a valores numéricos e estas cadeias de caracteres eram chamadas de *variáveis*. Com isto queríamos admitir que o processador fosse capaz de executar instruções do tipo *chame de a, b e c os coeficientes*. Como o nosso processador, de agora em diante, será um sistema de computação, as coisas têm que ser mais rigorosas.

Na seção 1.1 foi dito que uma das unidades básicas de um computador é a *memória*, cuja finalidade é armazenar dados e informações que serão manipulados pela *unidade central de processamento*; na seção 1.10 foi dito que os programas para serem executados devem também estar armazenados na memória. Ou seja, a memória armazena programas que serão executados e dados que estes programas vão manipular. Estes dados podem ser *dados de entrada* ou dados gerados pela execução do programa.

Para que a memória possa armazenar dados, ela é dividida em partes, chamadas *posições de memória*. O sistema operacional que gerencia o sistema de computação pode acessar cada uma destas posições para armazenar tais dados. Para que isto seja possível, a cada uma posição de memória está associada uma seqüência de bit's, chamada *endereço* da posição de memória. Como uma seqüência de bits corresponde a um número inteiro escrito no sistema binário, cada endereço pode ser visto como um inteiro escrito no sistema decimal. Assim temos posições de memória de endereço 1209 ou 2114, por exemplo.

Em programação, uma *variável simples* (ou simplesmente *variável*) é uma posição de memória cujo conteúdo pode ser modificado durante a execução de um programa, devendo ser-lhe associados um *identificador* e um *tipo de dado* .

O *identificador* é uma seqüência de letras, dígitos e caractere para sublinhamento escolhida pelo programador e será utilizado no programa para se fazer referência àquela variável (o primeiro caractere do identificador não pode ser um dígito). Como um programa deve ser legível por outros programadores (e pelo próprio programador), é uma boa prática se escolher um identificador de uma variável que tenha alguma relação com a sua finalidade. Se uma variável deve armazenar uma soma, um identificador muito bom para ela será *Soma*; se uma variável vai receber números, ela poderia ser identificada por *Num* ou por *Numero*. Os compiladores da linguagem Pascal não fazem distinção entre letras maiúsculas e minúsculas e, portanto, *Numero* e *numero* são identificadores idênticos e não podem ser utilizados para identificar variáveis distintas.

A linguagem Pascal fixa alguns identificadores para a sintaxe de suas instruções. Estes identificadores não podem ser utilizados nos programas, sendo conhecidos por *palavras reservadas*. A tabela a seguir apresenta algumas destas palavras reservadas.

Tabela 2 Palavras reservadas da linguagem Pascal

and	downto	In	or	then
asm	else	Inline	packed	to
array	end	Interface	procedure	type
begin	exports	Label	program	unit
case	file	Library	record	until
const	for	Mod	repeat	uses
constructor	function	Nil	set	var
destructor	goto	Not	shl	while
div	if	Object	shr	with
do	implementation	Of	string	xor

O *tipo de dado* associado a uma variável é um conjunto cujos elementos podem ser nela armazenados. A linguagem Pascal dispõe dos tipos de dados discriminados na tabela a seguir.

Tabela 3 Tipos de dados da Pascal

Denominação	Conjunto de valores
Char	caracteres codificados no código ASCII (letras, dígitos e caracteres especiais como :, ?, *, etc.)
Shortint	números inteiros do intervalo [-128, 127]
Integer	números inteiros do intervalo [–32768, 32767]
Longint	números inteiros do intervalo [-2 147 483 648, 2 147 483 647]
Byte	números inteiros do intervalo [0, 255]
Word	números inteiros do intervalo [0, 65535]
Real	números reais do conjunto $[-3{,}4\text{x}10^{38}, -3{,}4\text{x}10^{-38}] \cup [3{,}4\text{x}10^{-38}, 3{,}4\text{x}10^{38}]$
Boolean	conjunto dos valores *false* (falso) e *true* (verdadeiro)

À exceção do tipo *real*, os tipos de dados acima são *ordenados* no sentido de que existe um primeiro elemento e existem as idéias de sucessor e de antecessor. A ordenação dos tipos de dados que são subconjuntos dos inteiros é a ordenação natural da matemática; a do tipo de dado *char* é dada pelo Código ASCII e a do tipo de dado *boolean* é dada por {*false*, *true*}. Uma observação importante é que o tipo *real*, rigorosamente falando, não armazena números reais e sim números de um *sistema de ponto flutuante*, que não contém todos os reais entre dois números dados. O estudo de sistemas de ponto flutuante foge ao escopo deste livro e é feito em disciplinas do tipo *Organização e Arquitetura de Computadores* e *Cálculo Numérico*.

A utilização, quando possível, de um dos tipos *byte*, *shortint*, *integer* e *word* é ditada pela necessidade de economia de memória, já que variáveis do tipo *byte* e *shortint* requerem apenas um byte de memória, enquanto que variáveis dos tipos *integer* e *word* requerem dois bytes. Assim, se uma variável deve armazenar números inteiros pequenos, a ela deve ser associado o tipo *byte* ou o tipo *shortint*.

Para que o sistema de computação possa reservar as posições de memória que serão utilizadas pelo programa, associar identificadores aos endereços destas posições e definir a quantidade de bytes de cada posição de acordo com o tipo de dado pretendido, um programa escrito em Pascal deverá conter a *declaração de variáveis* feita através da seguinte sintaxe:

var Lista de identificadores: tipo de dado;

Por exemplo, um programa para determinar a média de uma relação de números dados pode ter a seguinte declaração:

```
var Quant: integer;
    Num, Soma, Media: real;
```

A idéia é que *Quant* seja utilizada para armazenar a quantidade de números; *Num* para armazenar os números (um de cada vez); *Soma* para armazenar a soma dos números; e *Media* para armazenar a média procurada.

Nas seções 2.6 e 2.8, veremos as instruções em Pascal para o armazenamento em variáveis de dados de entrada e de dados gerados pela execução do algoritmo. Um valor armazenado em uma variável é comumente referido como sendo o *conteúdo* ou o *valor* da variável. Também é comum se referir ao identificador da variável como sendo a própria variável.

2.2 Constantes

Uma *constante* é uma posição de memória na qual o sistema armazena um valor fixado pelo programa, valor este que não pode ser alterado durante sua execução. A uma constante é associado um identificador e qualquer referência posterior a este identificador será substituída pelo tal valor fixado. As constantes são declaradas com a seguinte sintaxe:

const identificador = valor;

Por exemplo, um programa para processar cálculos químicos poderia ter uma declaração do tipo

const NumAvogadro = 6.023E+23;

sendo 6.023E+23 a forma utilizada pelo sistema para escrever números reais na notação científica. Isto significa que 6.023E+23 = $6{,}023 \times 10^{23}$.

2.3 Expressões aritméticas

Os compiladores da linguagem Pascal, como era de se esperar, são capazes de avaliar expressões aritméticas que envolvam as operações binárias de multiplicação, divisão, soma e subtração e a operação unária de *troca de sinal*. Para isto são usados os *operadores aritméticos binários*

Tabela 4 Operadores aritméticos

Operador	Operação
+	adição
-	subtração
*	multiplicação
/	divisão

e o *operador aritmético unário* **-** para a troca de sinal. Os operadores +, -, * atuam com operandos dos tipos *integer* ou *real* fornecendo resultado do tipo *real* se pelo menos um dos operandos é do tipo *real* e resultado do tipo *integer* se ambos os operandos são deste tipo. O operador / sempre fornece resultados do tipo *real*. Isto significa que o resultado de uma expressão como 6/2 não pode ser armazenado numa variável do tipo *integer*.

Além destes, o sistema oferece dois outros operadores aritméticos que operam apenas com valores do tipo *integer* resultando valores também deste tipo. São os operadores *div* e *mod* que fornecem, respectivamente, o quociente e o resto da divisão inteira do primeiro operando pelo segundo. Por exemplo, 30 **div** 7 = 4 e 30 **mod** 7 = 2, enquanto que 5 **div** 7 = 0 e 5 **mod** 7 = 5. Como os operandos devem ser do tipo *integer*, uma expressão do tipo 30.0 **div** 7 gerará um erro de compilação.

Na avaliação de expressões, o sistema efetua primeiro as operações envolvendo *div*, *mod*, *, /, para depois efetuar as operações envolvendo (+) e (-). Isto é chamado de *prioridade* dos operadores. Observe que *div*, *mod*, (*) e (/) têm a mesma prioridade e que esta é maior do que a prioridade de (+) e (-). Como em matemática, a prioridade pode ser alterada com a utilização de parênteses. Se uma expressão envolvendo operadores de igual prioridade não for parentetizada, o sistema a avalia da esquerda para direita. Por exemplo, 60/4*3 = 15*3 = 45 enquanto que 60/(4*3) = 60/12 = 5.

Um detalhe mais ou menos evidente é que operandos podem ser conteúdos de variáveis. Neste caso, o operando é indicado pelo *identificador* da variável (é para isto que serve o identificador, para se fazer referência aos valores que na variável estão armazenados).

2.4 Relações

Os compiladores da linguagem Pascal realizam *comparações* entre valores numéricos, realizadas no sentido usual da matemática, e entre cadeias de caracteres, realizadas de acordo com a ordenação do código ASCII. Estas comparações são chamadas *relações* e são obtidas através dos *operadores relacionais*:

Tabela 5 Operadores relacionais

Operador	Operação
>	maior do que
>=	maior do que ou igual a
<	menor do que
<=	menor do que ou igual a
=	igual
<>	diferente

O resultado da avaliação de uma relação é *true*, se a relação for verdadeira, ou *false* se a relação for falsa. Assim, 3 > 5 resulta no valor *false*, enquanto que 7 <= 7 resulta no valor *true*. Sendo um valor *true* ou *false*, o resultado da avaliação de uma relação pode ser armazenado numa variável do tipo *boolean*.

Os operandos de uma relação podem ser expressões aritméticas. Nestes casos, as expressões aritméticas são avaliadas em primeiro lugar para, em seguida, ser avaliada a relação. Por exemplo, a relação 3*4 - 5 < 2*3 - 4 resulta no valor *false*, pois 3*4 - 5 = 7 e 2*3 - 4 = 2. Isto significa que os operadores relacionais têm prioridade mais baixa que os aritméticos.

2.5 Expressões lógicas

Os compiladores da linguagem Pascal também avaliam *expressões lógicas* obtidas através da aplicação dos *operadores lógicos binários and* e *or* a duas relações ou da aplicação do *operador lógico unário not* a uma relação.

Se r_1 e r_2 são duas relações, a avaliação da aplicação dos operadores lógicos binários, de acordo com os valores de r_1 e r_2, são dados na tabela abaixo.

Tabela 6 Avaliação de expressões lógicas com os operadores and e or

r_1	r_2	(r_1) **and** (r_2)	(r_1) **or** (r_2)
true	*true*	*true*	*true*
true	*false*	*false*	*true*
false	*true*	*false*	*true*
false	*false*	*false*	*false*

Uma expressão lógica do tipo *(r_1) and (r_2)* só recebe o valor *true* se os valores de r_1 e de r_2 forem iguais a *true*; uma expressão lógica do tipo *(r_1) or (r_2)* só recebe o valor *false* se os valores de r_1 e de r_2 forem iguais a *false*.

A aplicação do operador unário *not* simplesmente inverte o valor original da relação:

Tabela 7 Operador unário not

r_1	**not** r_1
true	*false*
false	*true*

Considerando que os operadores *and* e *or* possuem o mesmo grau de prioridade, se uma expressão não parentetizada possuir mais de uma relação, ela será avaliada da esquerda para direita. O operador unário *not* tem prioridade em relação aos operadores binários. Assim, *not* (5 > 3) *or* (5 < 3) tem valor *false*, pois *not* (5 > 3) é uma relação falsa e 5 < 3 também é. Como os operadores relacionais têm prioridade mais baixa que os operadores lógicos, os parênteses nas expressões acima são necessários.

2.6 Estrutura de um programa em Pascal

Estamos aprendendo a escrever programas na linguagem Pascal. Já vimos que se o programa necessitar manipular variáveis, estas devem ser declaradas. Vimos também que se pode definir constantes. Aprenderemos agora que um programa em Pascal contém *áreas de programa* distintas, cada uma delas com finalidade específica. A última das áreas de programa é a única que é obrigatória, é chamada *programa principal* e contém as instruções do programa propriamente dito e as *ativações* ou *chamadas* de *procedimentos* e de *funções*. O programa principal deve ser iniciado com a palavra *begin* (início, em inglês) e concluído com a palavra *end* (fim, em inglês), seguida de um ponto final. Desta forma, o "conjunto de instruções"

```
begin
end.
```

é um programa em Pascal (o menor programa em Pascal possível) que nada realiza, pois não contém nenhuma instrução propriamente dita. Os termos *begin* e *end* são chamados *delimitadores* e como veremos adiante são utilizados em muitas outras partes de um programa.

De um modo geral, as áreas de um programa em Pascal são:

identificação do programa,

relação das unidades utilizadas,

definições de tipos de dados,

declaração de constantes,

declaração de variáveis,

definições dos procedimentos e funções e

programa principal.

Em seguida discutiremos apenas a *identificação do programa*, a *relação das unidades utilizadas*, as *definições de tipos de dados* e as *definições de procedimentos e funções* já que as *declarações de constantes* e *de variáveis* e o *programa principal* já foram matéria de comentários.

A identificação do programa começa pela palavra obrigatória *program* seguida de um *identificador*, devendo o identificador seguir as mesmas regras de identificação de variáveis. Complementando o que já foi dito na seção 2.1, dois identificadores situados numa mesma área do programa devem ser distintos, distinção esta que deve ocorrer até 63º (sexagésimo terceiro) caractere. A identificação do programa é concluída com a aposição de um **;** (ponto e vírgula) após o identificador.

Uma *unidade* é um programa dos compiladores Pascal que contém vários *procedimentos e funções* pré-definidos, cujas utilizações facilitam muitas tarefas de programação. Por exemplo, o sistema possui a unidade *Crt* que contém o procedimento *ClrScr* cuja execução faz com que a *tela do usuário* seja limpa. Usualmente os compiladores Pascal oferecem uma tela para edição, chamada *tela de edição*, e uma tela pela qual são fornecidos os dados de entrada e os resultados do processamento são exibidos. Esta é a *tela do usuário* ou *tela de trabalho* (para se abrir a tela do usuário a partir da tela de edição deve se digitar <alt> + F5; para se retornar para a tela de edição, basta se digitar qualquer tecla).

Para que um programa possa utilizar um procedimento que pertence a uma determinada unidade, é necessário que a tal unidade seja relacionada no programa, isto sendo feito através da seguinte sintaxe:

uses lista das unidades;

Por exemplo,

```
program LimpaTela;
uses Crt;
begin
    ClrScr;
end.
```

é um programa que limpa a tela do usuário eventualmente utilizada por uma anterior execução de um outro programa.

Além da unidade *Crt*, os compiladores Pascal, de um modo geral, possuem as unidades *Dos*, *Graph*, *Printer* e *System*. A unidade *System* contém procedimentos e funções básicas de programação e não há necessidade de ser incluída na lista de unidades, pois ela é "carregada" na memória juntamente com o sistema. A unidade *Dos* contém procedimento e funções que permitem ao compilador Pascal interagir com o sistema operacional *DOS*; a unidade *Graph* contém procedimentos que permitem incluir gráficos nos programas e a unidade *Printer* permite interações do sistema com impressoras. Há uma outra unidade, denominada *Overlay*, que contém procedimentos e funções com objetivos que este livro não pretende alcançar.

Afora os tipos de dados do sistema, como explicado na seção 2.1, é possível definir novos tipos através da seguinte sintaxe:

type Identificador = caracterização do tipo de dado;

Por exemplo, um programa que manipula dias da semana pode ter uma definição de um tipo de dado como:

type Dias = (seg, ter, qua, qui, sex, sab, dom);

As formas para a caracterização de um tipo de dado que se está definindo serão estudadas posteriormente.

Além de definições de tipos definidos pelo usuário, esta área também é utilizada para associar identificadores a tipos pré-definidos. Isto é importante quando o tipo pré-definido é *estruturado* e vai ser utilizado como parâmetros de *procedimentos* ou e *funções*.

Os *procedimentos* e as *funções* são conjuntos de definições, declarações e comandos (a partir deste ponto, utilizaremos *comando* como sinônimo de *instrução*), com estrutura idêntica à estrutura de um

programa, sendo também chamados de *subprogramas* ou *subrotinas*. Estes elementos serão estudados no capítulo 5 e lá veremos que a utilização de procedimentos e funções facilita a tarefa de programação e permite que programas sejam de mais fácil compreensão através de leitura (a facilidade de compreensão de um programa através de sua leitura é normalmente referida como *legibilidade* do programa).

2.7 Entrada dos dados de entrada

Vimos anteriormente que a maioria dos programas manipula dados que são fornecidos pelo usuário durante a execução do programa. Estes dados, repetindo, constituem a *entrada do programa* e devem ser armazenados em variáveis. Por exemplo, um programa para determinação das raízes de uma equação do segundo grau deve receber como entrada os valores dos três coeficientes da equação, valores identificam a equação.

A entrada de dados é feita em Pascal através do *comando de entrada* que deve ser escrito com a sintaxe

readln(Lista de identificadores);

em que os identificadores são separados por vírgulas.

Quando da execução de um comando de entrada o processamento é interrompido, a *tela de edição* é substituída pela *tela do usuário* e o sistema fica aguardando que o usuário digite um número de valores (possivelmente seguidos da digitação da tecla <enter> ou da barra de espaços) igual ao número de variáveis da lista de variáveis (à medida que são digitados, os valores aparecem na tela do usuário). A conclusão da entrada dos dados é feita com a digitação da tecla <enter> e quando isto é feito o sistema armazena os dados digitados na variável respectiva, no sentido da ordem da colocação da variável na lista.

Por exemplo, o programa

```
program LeAno;
var Ano : integer;
begin
   readln(Ano);
end.
```

é um programa em Pascal que armazena na variável *Ano* um valor inteiro digitado no teclado (ou seja, para nada serve, pois o inteiro armazenado naquela posição de memória "evanesce" quando a execução do programa é encerrada).

É bom observar que, em alguns casos, a digitação de um valor de um tipo diferente do tipo da variável provoca erro de execução. Por exemplo, se na execução do "programa" anterior fosse digitado 3.25 ou *x*, ocorreria um erro de execução (a execução seria interrompida) e o sistema emitiria a mensagem:

Error 106: Invalid numeric format

Se a variável *Ano* tivesse sido declarada como *char* e na execução fosse dado como entrada o valor 234, não haveria erro de execução e o que seria armazenado seria o caractere 2.

2.8 Saída de dados

A exibição dos resultados do processamento e de mensagens é feita através dos *comandos de saída* de sintaxes

write(Lista de identificadores/Expressão/Mensagem);

ou

writeln(Lista de identificadores/Expressão/Mensagem);

em que as barras utilizadas têm o significado do conectivo e/ou da linguagem comum.

Quando um *argumento* de um comando de saída é um identificador de variável, a execução do comando redunda na exibição do conteúdo desta variável; quando é uma expressão, esta é avaliada e o seu resultado é exibido; quando é uma "mensagem" escrita entre apóstrofos, esta mensagem é exibida integralmente.

Por exemplo, o programa

```
program DeclaracaoImportante;
begin
   writeln('Estou aprendendo a programar em Pascal');
end.
```

exibe na tela a mensagem:

Estou aprendendo a programar em Pascal.

enquanto que o programa

```
program MediaDe3Versao1;
var a, b, c : real;
begin
   readln(a, b, c);
   write((a + b + c)/2);
end.
```

executado para a entrada 8, 6, 13 exibe na tela o valor 9.0000000000E+00, que é a média dos três números dados, escrita na notação científica.

Aproveitando o ensejo para dizer que os sistemas que implementam a linguagem Pascal disponibilizam uma constante pré-definida, identificada por *Pi*, cujo valor é o número irracional π, o programa abaixo fornece a área de um círculo de raio dado.

```
program AreaCirculo;
var Raio : real;
begin
   readln(Raio);
   write(Pi*Raio*Raio);
end.
```

A possibilidade de que mensagens possam ser exibidas permite que o próprio programa facilite a sua execução e que torne compreensíveis os resultados fornecidos. Da forma em que está escrito, a execução do programa acima que fornece a média de três números dados é dificultada pelo fato de que a execução do comando *readln* faz com que o sistema aguarde a digitação dos números pretendidos (o *cursor* fica simplesmente piscando na tela do usuário) e o usuário pode não saber o que está se passando. Além disto, a execução do comando *writeln* exibe apenas o resultado da expressão, sem indicação a que aquele valor se refere. Assim, o programa referido ficaria muito melhor da seguinte forma.

```
program MediaDe3Versao2;
var a, b, c : real;
begin
   write('Digite três números reais:  ');
   readln(a, b, c);
   writeln('A média dos números ', a,' , ', b , ' e ', c,  ' é ',  (a + b + c) / 3) );
end.
```

O primeiro comando *write* emite a mensagem "Digite três números reais:" indicando o que o usuário deve fazer para o programa continuar a ser executado, enquanto o segundo, além de fornecer a média esperada, indica o que aquele valor significa.

A diferença entre os comandos *write* e *writeln* é que o segundo ao final da exibição dos seus argumentos "exibe" o caractere de código ASCII 13, o que provoca uma mudança de linha. Assim, após a execução de um comando *writeln* o próximo caractere a aparecer na tela, via um comando de entrada ou via um comando de saída, será exibido na linha seguinte àquela que contém a exibição dos argumentos do referido comando *writeln*. Dizemos que o *cursor* (sinal intermitente, também chamado *ponto de inserção*, que indica a posição onde o próximo caractere a ser exibido, sê-lo-á,) *muda de linha*. O comando *writeln* inclusive pode não possuir argumentos. Neste caso, sua execução provoca apenas mudança de linha do cursor.

Por exemplo, o programa acima executado para a entrada 8.43, 6.1, 13.145, sendo a entrada de cada um deles separada por espaços, geraria a seguinte tela:

Digite três números reais: 8.43 6.1 13.145
A média dos números 8.4300000000E+00, 6.1000000000E+00 e 13.1450000000 é 9.2250000000E+00

enquanto que o programa

```
program MediaDe3Versao2;
var a, b, c : real;
begin
    write('Digite tres números reais:  ');
    readln(a, b, c);
    writeln;
    writeln;
    writeln('A média dos números ', a,' , ', b , ' e ', c,  ' é ',  (a + b + c) / 3) );
end.
```

executado com a mesma entrada, dada na mesma forma, geraria a tela:

Digite tres número reais: 8.43 6.1 13.145

A média dos números 8.4300000000E+00, 6.1000000000E+00 e 13.1450000000 é 9.2250000000E+00

Certamente o leitor já percebeu que estamos supondo que a primeira coluna da tela do computador coincide com a margem esquerda do livro.

Como os exemplos anteriores mostram, o padrão utilizado pelos compiladores Pascal é exibir os números de ponto flutuante na forma de notação científica, utilizando 10 casas decimais. Um outro padrão adotado é exibir o que deve ser exibido a partir da posição do cursor. Estes padrões podem ser alterados acrescentando-se parâmetros aos identificadores de variáveis, às expressões ou às mensagens que são argumentos de comandos de saída. Num comando de saída, após um identificador de variável, ou de uma expressão, ou de uma mensagem, pode-se acrescentar :n, definindo que o valor daquele argumento será exibido utilizando-se *n* colunas, a partir da posição do cursor. Por exemplo, se o conteúdo de uma variável A, do tipo *integer***,** é 56, a execução do comando

```
write(A);
```

exibe na tela o número 56 a partir da primeira coluna:

56

enquanto que a execução do comando

```
write(A:20)
```

exibe na tela o número 56 a partir da 19ª coluna:

                  56

De modo semelhante, o comando

```
write('Estou aprendendo a programar em Pascal');
```

exibe na tela a mensagem dada a partir da primeira coluna:

Estou aprendendo a programar em Pascal

enquanto que o comando

```
write('Estou aprendendo a programar em Pascal':60);
```

exibe a mensagem a partir da coluna 23 (já que a mensagem possui 38 caracteres):

                      Estou aprendendo a programar em Pascal

Na hipótese de valores do tipo *real*, pode-se acrescentar um segundo parâmetro :d para definir que o número de ponto flutuante seja exibido como número decimal com *d* casas decimais. Por exemplo, se o conteúdo de uma variável *Area*, do tipo *real***,** é 58.2362 o comando

```
write(A);
```

exibe na tela o valor 5.8236200000E+01; o comando

```
write(A:8);
```

exibe na tela o valor 5.82E+01; e o comando

```
write(A:8:2);
```

exibe 58.24, arredondado para duas casas.

A utilização dos dois parâmetros permite que se alinhe à direita a exibição de números reais (valores monetários, por exemplo). Se os conteúdos das variáveis do tipo *real a*, *b* e *c* são 1.235, 567.891 e 42.4589, a seqüência de instruções

```
writeln(a:20:2);
writeln;
writeln(b:20:2);
writeln;
writeln(c:20:2);
```

exibe na tela

```
  1.25

567.89

 42.46
```

2.9 Comando de atribuição

A seção 2.6 apresentou o comando que permite que os dados de entrada sejam armazenados em variáveis. Agora veremos como armazenar dados gerados durante a execução de um programa. Enfatizando o que foi dito na seção 1.7 relativo ao fato de que algoritmos podem manipular dados gerados por execuções de instruções anteriores, considere um programa para o cálculo da média de uma relação de números. Naturalmente a quantidade de números da relação (se não foi fornecida a priori) deve ser de alguma forma determinada e armazenada em alguma variável para que possa ser utilizada no cálculo final da média pretendida.

O armazenamento de dados gerados pelo próprio programa, alterações no conteúdo de variáveis e determinações de resultados finais de um processamento é feito através do *comando de atribuição* que deve ser escrito com a seguinte sintaxe.

Identificador de variável := expressão;

A expressão do segundo membro pode se resumir a um valor constante pertencente ao tipo de dado da variável do primeiro membro, caso em que o valor é armazenado naquela variável. Se não for este o caso, a expressão é avaliada e, se for do mesmo tipo da variável do primeiro membro, o resultado é nela armazenado.

Por exemplo, se *a* e *b* são variáveis do tipo *real*, a seqüência de comandos

```
a := 2.8;
b := a*a/2;
```

armazenará em *a* o valor 2.8 e em *b* o valor 3.92.

Outro detalhe interessante é que a expressão do segundo membro pode envolver a própria variável do primeiro membro. Neste caso, o conteúdo anterior da variável será utilizado para a avaliação da expressão e será substituído pelo valor desta expressão. Por exemplo, se *i* é uma variável do tipo *integer* ou do tipo *real* o comando

```
i := i + 1;
```

faz com que o seu conteúdo seja incrementado de uma unidade. Veremos que comandos deste tipo são muito comuns em programação.

Além da possibilidade de se dar entrada através do comando *readln*, pode-se dar entrada em caracteres utilizando-se a *função pré-definida ReadKey*. Para isto, deve-se atribuir a uma variável do tipo *char* a função citada e esta atribuição será executada quando o usuário digitar alguma tecla. Quando isto é feito, o caractere correspondente à tecla digitada é armazenado na variável do primeiro membro do comando de atribuição.

Por exemplo, a execução do programa

```
uses Crt;
var c: char;
begin
    writeln('Digite um caractere');
    c := ReadKey;
    writeln(' Voce digitou ', c, '.');
end.
```

com a digitação da letra A deixa a tela de trabalho da seguinte forma

Digite um caractere

Voce digitou a letra A.

Vale a pena notar que a tecla digitada não aparece na tela de trabalho como quando se dá entrada num dado através do comando *readln*. Por exemplo, na execução do programa

```
var c: char;
begin
    writeln('Digite um caractere');
    readln(c);
    writeln(' Voce digitou ', c, '.');
end.
```

a digitação do caractere A deixa a tela da seguinte forma:

Digite um caractere

A
Voce digitou a letra A.

Como a função *ReadKey* pertence à unidade *Crt*, é indispensável a instrução *uses Crt*; na primeira versão deste exemplo. Se esta unidade não for relacionada no programa, o compilador não reconhecerá a função pretendida e haverá um erro de compilação.

2.10 Exemplos Parte I

1. Voltando ao programa do cálculo da média de três números dados, observe que a média foi calculada e exibida, mas não foi armazenada. Se este programa fizesse parte de um programa maior (e isto normalmente acontece! Não se usa computação para uma questão tão simples!) e esta média fosse necessária em outra parte do programa, aquele trecho teria que ser reescrito. É uma boa prática, portanto, que resultados finais de processamento sejam armazenados em variáveis, sendo então os conteúdos destas variáveis exibidos através do comando *writeln*. Assim, o programa referido ficaria melhor escrito da seguinte forma.

```
{Programa que determina a média de três números dados}
program MediaDe3Versao3;
var a, b, c, Media : real;
begin
    writeln('Digite três números reais');
    readln(a, b, c);
    Media := (a + b + c)/3;
    writeln('A média dos números ', a,' , ', b ,' e ', c,' é ', Media);
end.
```

2. Agora apresentaremos um programa que, recebendo um número inteiro como entrada, fornece o algarismo da casa das unidades deste número, questão discutida na seção 1.6. Como vimos naquela seção, o

algarismo procurado é o resto da divisão do número dado por 10. Temos então o seguinte programa (no capítulo 6 veremos um programa que necessita da solução desta questão):

```
{Programa que determina o algarismo da casa das unidades de um inteiro dado}
program AlgarismoDasUnidades;
var n, Unid : integer;
begin
    writeln('Digite um inteiro');
    readln(n);
    Unid := n mod 10;
    writeln('O algarismo das unidades de  ', n, ' é ', Unid);
end.
```

3. Se quiséssemos um programa para inverter um número com dois algarismos (por exemplo, se a entrada fosse 74, a saída deveria ser 47), poderíamos utilizar o seguinte fato: se *x* e *y* são os algarismos de um número (casa das dezenas e das unidades, respectivamente), então este número é x . 10 + y. Assim, a inversão seria y . 10 + x (no exemplo, 74 = 7 . 10 + 4; 47 = 4 . 10 + 7). Desta forma, basta extrair os dois algarismos do número dado e utilizar a expressão acima. A extração do algarismo da casa das unidades foi mostrada no exemplo anterior. E o algarismo da casa das dezenas? Basta ver que ele é o quociente da divisão do número por 10, podendo este quociente ser obtido através do operador *div*. Temos então o seguinte programa:

```
{Programa que inverte um número com dois algarismos}
program InverteNumeroComDoisAlgarismos;
var n, Inv, Unid, Dez : integer;
begin
    writeln('Digite um inteiro com dois algarismos');
    readln(n);
    Unid := n mod 10;
    Dez := n div 10;
    Inv := Unid * 10 + Dez;
    writeln('O invertido de ', n,  ' é ', Inv );
end.
```

Dificilmente o caro leitor vai escrever um programa com este objetivo (para que serve inverter um número com dois algarismos?). Esta questão, e algumas outras, estão sendo discutidas aqui apenas como exemplos para o desenvolvimento da lógica de programação e pelo fato de que podem ser trechos de programas maiores.

4. Imagine agora que queiramos um programa que determine o maior múltiplo de um inteiro dado menor do que ou igual a um outro inteiro dado. Por exemplo, se a entrada fosse 13 e 100, a saída deveria ser 91 (91 é o maior múltiplo de 13 que é menor do que ou igual a 100).

Como *dividendo = divisor x quociente + resto*, temos que o valor da expressão *dividendo – resto* é o múltiplo procurado.

```
{Programa que determina o maior múltiplo de um inteiro inferior ou igual a outro inteiro}
program MaiorMultiplo;
var n, k, MaiorMult : integer;
begin
    writeln('Digite dois inteiros ');
    readln(n, k);
    MaiorMult := n -  n mod k;
    writeln('O maior múltiplo de ', k, ' inferior ou igual a ', k, ' é ', MaiorMult);
end.
```

5. O programa a seguir, além de ser muito interessante no sentido do desenvolvimento da lógica de programação, será utilizado (a seqüência de comandos do programa principal) em outros programas. O objetivo dele é permutar os conteúdos de duas variáveis. Ou seja, suponhamos que, através de comandos de entrada, o programa armazenou, nas variáveis *x* e *y*, os valores 7 e 18 e pretendamos que o programa faça com que o conteúdo de *x* passe a ser 18 e o de *y* passe a ser igual a 7. À primeira vista, bastaria a seqüência

de comandos

```
x := y
y := x
```

Ocorre que, quando o segundo comando fosse executado, o primeiro já teria sido e o conteúdo de *x* não seria mais o original. No nosso exemplo, teríamos a seguinte situação:

x	y
17	8
8	
	8

e a permuta não teria sido feita, além do fato de que o conteúdo original de *x* teria sido perdido. Uma alternativa seria considerar duas variáveis que "guardem" os conteúdos de *x* e de *y*.. Teríamos assim o seguinte programa:

```
{Programa que permuta os conteúdos de duas variáveis}
program Troca;
var x, y, Xant, Yant : real;
begin
    readln(x, y);
    writeln('Entrada: x = ', x, ' e  y = ', y);
    Xant := x;
    Yant := y;
    x := Yant;
    y := Xant;
    writeln('Saida: x = ', x, ' e  y = ', y);
end.
```

O programa acima é eficaz no sentido de que ele realiza a tarefa proposta. Porém, na seção 1.9, foi dito que se deve procurar o *melhor programa* no sentido de que o programa demande o menor tempo de execução possível e que necessite do mínimo de memória. Em relação ao programa acima, não seria possível uma solução que utilizasse apenas uma variável? Basta ver que, se armazenarmos o conteúdo de *x* numa variável auxiliar, o conteúdo de *y* pode ser armazenado em *x* diretamente. Temos então a seguinte solução:

```
{Programa que permuta os conteúdos de duas variáveis, utilizando uma variável auxiliar}
program Troca;
var x, y, Aux : real;
begin
    readln(x, y);
    writeln('Entrada: x = ', x, ' e  y = ', y);
    Aux := x;
    x := y;
    y := Aux;
    writeln('Saida: x = ', x, ' e  y = ', y);
end.
```

Continuando a procura por um programa melhor, será que há solução sem utilizar uma variável auxiliar? Isto será deixado como exercício proposto.

2.11 Funções predefinidas

Os compiladores da linguagem Pascal oferecem diversas funções com objetivos predeterminados e que podem ser executadas durante a execução de um programa. Para isto a execução da função deve ser solicitada no programa como uma instrução, como operando de uma expressão ou como argumento de outra função (a solicitação da execução de uma função é normalmente chamada de *ativação* ou *chamada* da função). Para que o programador possa colocar no seu programa uma instrução que ative uma função, é necessário que o ele conheça o *identificador* da função, quantos e de que tipo são os *argumentos* com que ela deve ser ativada e o tipo de valor que ela *retorna* ao programa quando termina sua execução.

O quadro seguinte apresenta algumas destas funções, sendo que *ordenado* significa qualquer tipo ordenado (*integer*, *char*, *boolean*) e *mesmo* significa que o valor calculado será do mesmo tipo do argumento. Estas funções estão contidas na unidade *System* e são implementadas em linguagem de máquina. Como a unidade *System* é carregada na memória junto com o sistema, estas funções estão disponíveis para qualquer programa.

Tabela 8 Algumas funções predefinidas da linguagem Pascal

Função	Argumento	Valor	Valor calculado
Abs(x)	real/integer	mesmo	valor absoluto de x
ArcTan(x)	real	real	número cuja tangente é x
Chr(x)	byte	char	caractere cujo código ASCII é x
Cos(x)	real	real	coseno de x
Exp(x)	real	real	exponencial de x (e^x)
Frac(x)	real	real	parte fracionária de x
Ln(x)	real	real	logaritmo natural de x
Odd(x)	integer	boolean	verifica se x é ímpar
Ord(x)	tipo ordenado	integer	ordem de x no tipo de dado
Pred(x)	tipo ordenado	mesmo	antecessor de x
Round(x)	real	integer	arredondamento de x
Sin(x)	real	real	seno de x
Sqr(x)	real/integer	mesmo	quadrado de x
SqrT(x)	real/integer	real	raiz quadrada x
Succ(x)	tipo ordenado	mesmo	sucessor de x
Trunc(x)	real	integer	parte inteira de x
UpCase(x)	char	char	X maiúsculo

As denominações da maioria destas funções são as mais naturais possíveis. *Sqr* e *SqrT* advêm de *SQuaRe* (quadrado em inglês) e *SQuaRe rooT* (raiz quadrada em inglês). A palavra *odd* em inglês identifica os números ímpares.

2.12 Exemplos Parte II

Quando estudarmos as *funções* no capítulo 5, as soluções dos exemplos a seguir serão mais consistentes. Por enquanto vejamos alguns programas que poderiam *implementar* algumas das funções acima. Estes exemplos podem ser úteis quando o programador está utilizando um sistema que não oferece determinada função e oferece outras.

1. O primeiro exemplo apresenta um programa que implementa a função de implementação mais simples, no caso a função *Sqr*:

```
program ImplementaQuadrado;
var n, Quadrado : real;
begin
   writeln(´Digite um numero´);
   readln(n);
   Quadrado := n*n;
   writeln(´O quadrado de ´, n, ´ é ´, Quadrado);
end.
```

2. Utilizando algumas das funções predefinidas, podemos escrever programas que "implementam" outras funções. Repetindo o que foi dito acima, a idéia discutida num programa desse poderia ser utilizada numa linguagem que não oferecesse uma dada função e o programador tivesse necessidade de utilizá-la. Por exemplo, utilizando a função *Trunc* o programa abaixo "implementa" a função *Round*.

```
program Arredonda;
var x : real;
    Roundx: integer;
begin
   writeln('Digite um numero real');
```

```
        readln(x);
        Roundx := Trunc(2*x) - Trunc(x);
        writeln('Round(' , x , ') = ', Roundx);
    end.
```

Observe a idéia bastante intuitiva de subtrair a parte inteira do número dado da parte inteira do seu dobro. A regra do "vai um" captura o arredondamento quando a primeira casa decimal do número é maior do que ou igual 5.

Este exemplo é interessante no desenvolvimento da lógica de programação, pois ele permite outra solução também, de certa forma, engenhosa: se a parte fracionária de *x* é maior ou igual 0.5, acrescentando-se 0.5 a *x* a parte inteira de *x* é acrescida de uma unidade. Assim, no programa anterior o comando

```
Roundx := Trunc(2*x) - Trunc(x);
```

poderia ser substituído por

```
Roundx := Trunc(x + 0.5);
```

Além destas duas soluções, no próximo capítulo apresentaremos uma solução que utiliza diretamente o conceito de arredondamento.

3. Ainda na linha do exemplo anterior, o programa abaixo utiliza as funções *Chr* e *Ord* para "implementar" a função *UpCase*. Para compreendê-lo, precisamos saber que *Ord*('A') = 65, *Ord*('a') = 97 e que os valores de *Ord* para as demais letras são seqüenciais em relação à ordem alfabética.

```
program UpCase;
var Maiuscula, Minuscula : char;
begin
    writeln('Digite uma letra minúscula');
    readln(Minuscula);
    Maiuscula := Chr(Ord(Minuscula) - 32);
    writeln('UpCase(' , Minuscula , ') = ', Maiuscula);
end.
```

Observe que *Ord*(Minuscula) - 32 (32 vem de 97 - 65) determina o valor da função *Ord* para a maiúscula correspondente à minúscula dada e *Chr* determina então a maiúscula procurada. Uma questão que se põe é: e se o caractere digitado para o comando *readln*(Minuscula) não for uma letra minúscula? A resposta a esta questão será dada no capítulo seguinte.

4. Para exemplificar a situação de um programador que está utilizando um sistema que não possui uma função que ele necessita, podemos discutir o exemplo de uma função que os compiladores Pascal não oferecem. Trata-se de uma função "inversa" da função *UpCase*, que converta uma letra maiúscula numa minúscula. Para o leitor ter uma idéia, os compiladores da linguagem C disponibilizam a função *tolower()* que executa esta ação. No nosso caso, tendo entendido o programa acima, o programa abaixo se torna de fácil entendimento.

```
program LowCase;
var Maiuscula, Minuscula : char;
begin
    writeln('Digite uma letra maiúscula');
    readln(Maiuscula);
    Minuscula := Chr(Ord(Maiuscula) + 32);
    writeln('LowCase(' , Maiuscula , ') = ', Minuscula);
end.
```

Naturalmente, a questão levantada no exemplo anterior persiste de forma semelhante: e se a letra digitada não for uma letra maiúscula?

6. O exemplo a seguir, além de pretender motivar o próximo capítulo, objetiva também ressaltar algo mais ou menos óbvio, mas que deve ser destacado: um programador só é capaz de escrever um programa que resolva um determinado problema se ele souber resolver o tal problema "na mão", ou seja, com a utilização apenas de lápis e papel. Trata-se de um programa que calcule a área de um triângulo, dados os comprimentos dos seus lados. Naturalmente, só é capaz de escrever este programa aquele que conhecer a fórmula abaixo,

que dá a área do triângulo cujos lados têm comprimentos x, y e z.

$$S = \sqrt{p \cdot (p - x) \cdot (p - y) \cdot (p - z)}$$

onde $p = \frac{x + y + z}{2}$ é o *semiperímetro* do triângulo. O programa abaixo utiliza uma variável *SemiPer* para calcular o semiperímetro do triângulo facilitando a forma da expressão do cálculo da área. O armazenamento do valor do semiperímetro em uma variável também seria útil se esse dado fosse necessário em outro trecho do programa.

```
{Programa que determina a área de um triângulo de lados de comprimentos dados}
program AreaTriangulo;
var a, b, c, SemiPer, Area : real;
begin
   writeln('Digite os lados do triângulo')
   readln(a, b, c);
   SemiPer := (a + b + c)/2;
   Area := SqrT(SemiPer*(SemiPer - a)*(SemiPer - b)*(SemiPer - c));
   writeln('Área do triângulo =  ', Area);
end.
```

Se este programa for executado com entrada 3, 4 e 5, temos *SemiPer* = 6 e

$$Area = \sqrt{6 \cdot (6 - 3) \cdot (6 - 4) \cdot (6 - 5)} = \sqrt{36} = 6$$

e, como era de se esperar, a área do triângulo cujos lados têm comprimento 3, 4 e 5 unidades de comprimento é igual a 6 unidades de área.

Agora, se este programa fosse executado para entrada 1, 2 e 5, teríamos *SemiPer* = 4 e

$$Area = \sqrt{4 \cdot (4 - 1) \cdot (4 - 2) \cdot (4 - 5)} = \sqrt{-24}$$

e ocorreria erro de execução, pois o sistema, como era de se esperar, não calcula raiz quadrada de número negativo.

O que acontece é que nem sempre três números podem ser comprimentos dos lados de um triângulo (a matemática prova que isto só acontece se cada um deles for menor do que a soma dos outros dois). Assim, o comando que calcula a *Area* só deveria ser executado se os valores digitados para x, y, e z pudessem ser comprimentos dos lados de um triângulo.

2.13 Exercícios propostos

1. Avalie cada uma das expressões abaixo.

a) (-(-9) + **SqrT**((-9)*(-9) - 4*3*6))/(2*3).

b) ((Round(6,78) = 7) **and** Odd(78)) **or** (4 **mod** 8 = 4).

2. Escreva programas para

a) Converter uma temperatura dada em graus Fahrenheit para graus Celsius.

b) Gerar o *invertido* de um número com três algarismos (exemplo: o *invertido* de 498 é 894).

c) Somar duas frações ordinárias, fornecendo o resultado em forma de fração.

d) Determinar o menor múltiplo de um inteiro dado maior do que um outro inteiro dado (exemplo: o menor múltiplo de 7 maior que 50 é 56).

e) Determinar o perímetro de um polígono regular inscrito numa circunferência, dados o número de lados do polígono e o raio da circunferência.

3. Escreva um programa que permute o conteúdo de duas variáveis sem utilizar uma variável auxiliar (ver exemplo 5 da seção 2.9).

4. Uma loja vende seus produtos no sistema entrada mais duas prestações, sendo a entrada maior do que ou igual às duas prestações, as quais devem ser iguais, inteiras e as maiores possíveis. Por exemplo, se o valor da mercadoria for R$ 270,00, a entrada e as duas prestações são iguais a R$ 90,00; se o valor da mercadoria for R$ 302,75, a entrada é de R$ 102,75 e as duas prestações são a iguais a R$ 100,00. Escreva um programa que receba o valor da mercadoria e forneça o valor da entrada e das duas prestações, de acordo com as regras acima. Observe que uma justificativa para a adoção desta regra é que ela facilita a confecção e o conseqüente pagamento dos boletos das duas prestações.

5. Um intervalo de tempo pode ser dado em dias, horas, minutos, segundos ou seqüências "decrescentes" destas unidades (em dias e horas; em horas e minutos; em horas, minutos e segundos), de acordo com o interesse de quem o está manipulando. Escreva um programa que converta um intervalo de tempo dado em segundos, em horas, minutos e segundos. Por exemplo, se o tempo dado for 3 850 segundos, o programa deve fornecer 1 h 4 min 10 s.

6. Escreva um programa que converta um intervalo de tempo dado em minutos, em horas, minutos e segundos. Por exemplo, se o tempo dado for 145,87 min, o programa deve fornecer 2 h 25 min 52,2 s.

7. Um programa para gerenciar os saques de um caixa eletrônico deve possuir algum mecanismo para decidir o número de notas de cada valor que deve ser disponibilizado para o cliente que realizou o saque. Um possível critério seria o da "distribuição ótima" no sentido de que as notas de menor valor fossem distribuídas em número mínimo possível. Por exemplo, se a quantia solicitada fosse R$ 87,00, o programa deveria indicar uma nota de R$ 50,00, três notas de R$ 10,00, uma nota de R$ 5,00 e duas notas de R$ 1,00. Escreva um programa que receba o valor da quantia solicitada e retorne a distribuição das notas de acordo com o critério da distribuição ótima.

8. Os sistemas que implementam a linguagem Pascal não possuem função predefinida para calcular potências com expoentes maiores que dois. Utilize o fato de que $a^x = e^{xLn(a)}$ para escrever um programa que determine o valor de uma potência a^y, a e y reais dados, com a positivo.

9. De acordo com a Matemática Financeira, o cálculo das prestações para amortização de um financiamento de valor F em n prestações e a uma taxa de juros i é dada pela fórmula $P = F/a_{n\neg i}$, onde $a_{n\neg i} = ((1 + i)^n – 1)/(i . (1 + i)^n)$. Escreva um programa que determine o valor das prestações para amortização de um financiamento, dados o valor do financiamento, o número de prestações para amortização e a taxa de juros.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 3 Estruturas de Seleção

3.1 O que é uma estrutura de seleção

O último exemplo do capítulo anterior apresentava um programa para calcular a área de um triângulo, dados os comprimentos dos seus lados. Foi visto que o comando que calculava a área solicitada só deveria ser executado com a certeza anterior de que os valores dados como entrada poderiam ser comprimentos dos lados de um triângulo. Em outras palavras, o tal comando só deveria ser executado *se* $x < y + z$ *e* $y < x + z$ *e* $z < x + y$, condição que garante que os valores armazenados nas variáveis *x*, *y* e *z* são comprimentos dos lados de um triângulo. Assim, e reforçando o que foi dito na seção 1.6, existem situações em que alguns comandos só devem ser executados se alguma condição for verificada.

A verificação de que um condição é satisfeita e, a partir daí, a decisão em relação à execução ou não de uma determinada seqüência de comandos é chamada de *estrutura de seleção*, *estrutura de decisão* ou *comando de seleção*.

3.2 O comando *if then*

O comando *if then* é uma estrutura de decisão que define se uma seqüência de comandos será ou não executada. Sua sintaxe é

```
if  Expressão lógica
    then
        begin
           seqüência de comandos
        end;
```

sendo os delimitadores opcionais se a seqüência de comandos contém um único comando.

A semântica deste comando é muito simples: se o valor da *Expressão lógica* for *true*, o sistema executará a seqüência de comandos; caso contrário, o sistema não executará a seqüência de comandos e a instrução após o comando *if then* passa a ser executada.

3.3 Exemplos Parte III

1. Se queremos um programa que determine o maior de dois números dados, podemos supor que o primeiro deles é o maior, armazenando-o numa variável *Maior* e depois, através de um comando *if then*, verificar se o maior procurado é o segundo dos números dados; neste caso o conteúdo da variável *Maior* deve ser alterado.

```
{Programa para determinar o maior de dois números dados}
program MaiorDe2;
var a, b, Maior : real;
begin
    writeln('Digite dois números');
    readln(a, b);
    Maior := a;
    if  (b > a)
        then
            Maior =  b;
    writeln('O maior dos números ', a, ' e ', b, ' é ', Maior);
end.
```

2. Um outro exemplo de utilização do comando *if then* aparece num programa que pretenda ordenar os conteúdos de variáveis *x* e *y*. Para isto só há de necessidade de se realizar alguma ação se o conteúdo de *y* for maior do que o conteúdo de *x*. Neste caso, o que deve ser feito é a permuta dos conteúdos de *x* e de *y*. Temos

então o seguinte programa:

```
{Programa para ordenar os conteúdos de duas variáveis}
program Ordena2;
var x, y, Aux : real;
begin
    writeln('Digite os dois números ');
    readln(x, y);
    writeln('Números digitados: 'x = ', x:4:2,  ' y = ', y:4:2);
    if  x > y
        then
            begin
                Aux := x;
                x := y;
                y := Aux;
            end;
     writeln('Números ordenados: x = ', x:4:2, ' y = ', y:4:2);
end.
```

Observe que a seqüência de comandos

```
Aux :=x;
x := y;
y := Aux;
```

realiza a permuta dos conteúdos das variáveis x e y como discutido no exemplo 5 da seção 2.9.

Outra observação importante é que, como os delimitadores *begin* e *end* são opcionais quando a seqüência de comandos *vinculada* ao comando *if then* possui um único comando, a não colocação deles faz com que o sistema só vincule ao comando de decisão o primeiro comando da seqüência. O esquecimento de delimitadores necessários, fatalmente, trará problemas de lógica para o programa. Por exemplo, se executarmos o programa *Ordena2* acima para x = 8 e y = 5, teríamos:

x	y	Aux	exibe na tela
8	5		
			Números digitados: x = 8 y = 5
		8	
5			
	8		
			Números ordenados: x = 5 y = 8

Se os delimitadores, porém, forem esquecidos e escrevermos o programa acima da forma abaixo

```
program Ordena2;
var x, y, Aux : real;
begin
    writeln('Digite os dois números ');
    readln(x, y);
    writeln('Números digitados: 'x = ', x, '  y = ', y);
    if  x > y
        then
            Aux := x;
            x := y;
            y := Aux;
     writeln('Números ordenados: x = ', x:4:2, ', y = ', y:4:2);
end.
```

e o executarmos para x = 5 e y = 8, a expressão lógica que controla o *if* é falsa e então o comando *Aux* := *x*, não seria executado, mas os comandos *x* := *y* e *y* := *Aux* seriam e assim o conteúdo de *x* passaria a ser 8 e o conteúdo de y armazenaria o "conteúdo" de *Aux*, sobre os quais não se tem controle (alguns compiladores Pascal armazenam o valor 0 (zero) quando a variável é declarada. Este armazenamento inicial é chamado *inicialização* da variável).

É interessante notar que, mesmo com o erro de lógica, esta nova versão do programa funciona corretamente para entradas em que $x > y$.

O fato de termos escrito os comandos vinculados ao comando de seleção com uma tabulação "maior" será comentado na seção 3.6.

3. Considere agora a questão de se determinar o número de anos bissextos entre dois anos dados. Sabe-se que um ano é bissexto se ele é múltiplo de quatro, exceto aqueles que são múltiplos de 100, mas não são múltiplos de 400 (por exemplo, 2012, múltiplo de 4, será bissexto; 1900, múltiplo de 100, mas não de 400, não foi bissexto; 2000, múltiplo de 100 e de 400, foi bissexto). Assim a determinação do número de anos bissextos entre dois anos dados consiste em se determinar o número de múltiplos de 4, *Multiplos4*, o número de múltiplos de 100, *Multiplos100*, e o número de múltiplos de 400, *Multiplos400*, todos compreendidos entre os anos dados, e se calcular a expressão *Multiplos4 – Multiplos100 + Multiplos400*.

Para se determinar o número de múltiplos de 4 (e *mutatis mutandis* os números dos múltiplos de 100 e de 400) situados entre dois anos *Ano1* e *Ano2*, basta calcular a expressão *(Ano1 div 4) - (Ano2 div 4)*, diminuída de uma unidade quando *Ano2* for múltiplo de 4. Temos então o seguinte programa, no qual só há diminuição referida quando é necessário:

```
{Programa para determinar o numero de anos bissextos entre dois anos}
program NumeroAnosBissextos;
var Ano1, Ano2, AnBiss, Multiplos4, Multiplos100, Multiplos400 : integer;
begin
    writeln('Digite os dois anos');
    readln(Ano1, Ano2);
    Multiplos4 := Ano2 div 4 - Ano1 div 4;
    if Ano2 mod 4 = 0
        then
            Multiplos4 := Multiplos4 - 1;
    Multiplos100 := Ano2 div 100 - Ano1 div 100;
    if Ano2 mod 100 = 0
        then Multiplos100 := Multiplos100 - 1;
    Multiplos400 := Ano2 div 400 - Ano1 div 400;
    if Ano2 mod 400 = 0
        then Multiplos400 := Multiplos400 - 1;
    AnBiss := Multiplos4 - Multiplos100 + Multiplos400;
    writeln('Numero de anos bissextos entre os anos ', Ano1, ' e ', Ano2, ': ', AnBiss);
end.
```

3.4 O comando *if then else*

O comando *if then else* é uma estrutura de decisão que decide entre duas seqüências de comandos qual vai ser executada, sendo definido através da seguinte sintaxe:

```
if  Expressão lógica
   then
        begin
            seqüência de comandos 1;
         end
   else
        begin
            seqüência de comandos 2;
         end;
```

A semântica deste comando é a seguinte: se o valor de *Expressão lógica* for *true*, o sistema executará a seqüência de comandos 1; caso contrário, o sistema executará a seqüência de comandos 2.

3.5 Exemplos Parte IV

1. No último exemplo do capítulo 2, apresentamos um programa que calculava a área de um triângulo, dados os comprimentos dos seus lados. No final dele, mostramos que o mesmo não fornecia respostas satisfatórias para todas as entradas e comentamos que o cálculo da área deveria ser precedido da verificação de que os dados de entrada são de fato comprimentos dos lados de um triângulo. O programa referido escrito agora de forma completa e correta seria o seguinte.

```
{Programa para calcular a area de um triangulo}
program AreaTriangulo;
var x, y, z, Area, SemiPer : real;
begin
    writeln('Digite os comprimentos dos lados do triangulo');
    readln(x, y, z);
    if (x < y + z) and (y < x + z) and (z < x + y)
        then
            begin
                SemiPer := (x + y + z)/2;
                Area := SqrT(SemiPer*(SemiPer - x)*(SemiPer - y)*(SemiPer - z));
                writeln('Area do triangulo de lados ',x, ', ', y,' e ', z,': ', Area:0:2);
            end
        else
            writeln(x,', ', y,' e ', z,' não podem ser comprimentos dos lados de um triangulo ');
end.
```

2. Se queremos um programa que verifique a paridade de um número *n* dado, podemos determinar o valor de *Odd*(n). Se este valor for *true*, o número é ímpar; caso contrário, o número dado é par.

```
{Programa para verificar a paridade de um número}
program VerificaParidade;
var n : integer;
begin
    writeln('Digite um numero inteiro');
    readln(n);
    if Odd(n) = true
        then writeln(n,' e impar')
        else  writeln(n,' e par');
end.
```

Vale observar que o valor de *Odd*(n) é um valor do tipo *boolean*: *true* ou *false*. Assim, como a seqüência vinculada à opção *then* será executada se a expressão *Odd*(n) = *true* for *true* e isto, evidentemente, só é verdade se este for o valor de *Odd*(n), o comando *if* acima poderia ser escrito simplesmente:

```
if Odd(n)
    then . . .
```

3. Foi alertado que o programa que "implementa" a função *UpCase*, apresentado no exemplo 3 da seção 2.11, fornecia resultados incorretos se o dado de entrada não fosse uma letra minúscula. Esta questão poderia se resolvida colocando-se uma estrutura de decisão que verificasse se o caractere digitado era realmente uma letra minúscula. Isto seria verdadeiro se *Ord*(Minuscula) ≥ 97 e *Ord*(Minuscula) ≤ 122, pois *Ord*('a') = 97 e *Ord*('z') = 122.

```
program UpCase;
var Maiuscula, Minuscula : char;
begin
    writeln('Digite uma tecla');
    readln(Minuscula);
    if (Ord(Minuscula) >= 97) and (Ord(Minuscula) <= 122)
        then
```

```
                Maiuscula := Chr(Ord(Minuscula) - 32)
            else
                Maiuscula := Minuscula;
        writeln(Maiuscula);
    end.
```

Observe que o caractere é recebido no comando de entrada e o programa verifica se se trata de uma letra minúscula. Em caso afirmativo, a variável *Maiuscula* recebe a maiúscula correspondente e, em caso negativo, *Maiuscula* recebe o próprio caractere de entrada, já que nada precisa ser feito.

4. O programa *Arredonda*, apresentado no exemplo 2 da seção 2.11, utilizava idéias, talvez mágicas, para se arredondar um número real dado utilizando a função predefinida *Trunc*. Utilizando o comando *if* e a função predefinida *Frac*, o programa abaixo, embora mais extenso, é mais natural para obter o mesmo efeito.

```
program Arredonda;
var     x : real;
Roundx : integer;
begin
        writeln('Digite um número real');
        readln(x);
        if Frac(x) < 0.5
            then
                Roundx := Trunc(x)
            else
                Roundx := Trunc(x) + 1;
        writeln('Round(' , x , ') = ', Roundx:4:2);
end.
```

5. Programas que manipulam datas (por exemplo, um programa que determine o número de dias entre duas datas dadas) contêm trechos que verificam se um ano dado é bissexto. Levando em conta o fato exposto no exemplo 3 da seção 3.3 de que um ano é bissexto quando ele é múltiplo de quatro, exceto aqueles que são múltiplos de 100, mas não são múltiplos de 400, o programa abaixo verifica se um ano dado é bissexto.

```
program Bissexto;
var Ano : integer;
begin
    writeln('Digite o ano ');
    readln(Ano);
    if (Ano mod 400 = 0) or ((Ano mod 4 = 0) and (Ano mod 100) <> 0)
        then
            write(Ano, ' é bissexto')
        else
            write(Ano, ' não é bissexto');
end.
```

Observe a expressão lógica escolhida para *controlar* o comando *if*: Todo ano múltiplo de quatrocentos (Ano mod 400 = 0) é bissexto; além destes (daí o operador lógico *or*), os anos múltiplos de 4 que não são múltiplos de 100.

6. O programa para ordenar os conteúdos de duas variáveis, visto na seção 3.3, é um caso muito particular da questão mais geral da *ordenação* de uma relação de números ou de nomes, problema que tem vasta aplicação na vida prática (principalmente na ordenação de uma lista de nomes); este problema também é conhecido como *classificação*. Para a solução geral existem diversos algoritmos com este objetivo e no capítulo 10 teremos oportunidade de discutir programas baseados em alguns destes algoritmos. Por enquanto, vejamos um programa que ordene três números dados. Além de exemplificar o comando *if then else*, o programa abaixo mostra como se pode (e se deve) utilizar raciocínios anteriores para se escrever programas.

Seja então um programa que receba três números inteiros, armazene-os nas variáveis *x*, *y* e *z* e que, ao final da sua execução, deixe os conteúdos de *x*, de *y* e de *z* na ordem crescente. Uma idéia bem interessante é armazenar na variável *x* o menor dos números e em seguida ordenar os conteúdos de *y* e de *z*, que é

exatamente o problema de ordenar os conteúdos de duas variáveis, que foi referido acima. Obviamente, para se executar a primeira ação pretendida (armazenar na variável x o menor dos números), só é necessário se fazer alguma coisa se o valor de x já não for o menor dos números dados, ou seja, se $x > y$ ou $x > z$. Nesta hipótese, o menor deles é y ou z e este menor deve ser permutado com x. Temos então o seguinte programa.

```
{Programa para ordenar três números dados}
program Ordena3;
var x, y, z, Aux: integer;
begin
    writeln('Digite os tres  números: ');
    readln(x, y, z);
    writeln('Números digitados: x = ', x, ' y = ', y, ' z = ', z);
    if (x > y) or (x > z)           {Se isto for verdade, x não é o menor}
        then
            if (z > y)                    {Se isto for verdade, y é o menor}
                then
                    begin                 {Permuta os conteúdos de x e de y}
                        Aux := x;
                        x := y;
                        y := Aux;
                    end
                else                {Neste caso, z é o menor}
                    begin                 {Permuta os conteúdos de x e de z}
                        Aux := x;
                        x := z;
                        z := Aux;
                    end;
    if (y > z)                            {Permuta os conteúdos de y e de z}
        then
            begin
                Aux := y;
                y := z;
                z := Aux;
            end;
    writeln('Números ordenados: x = ', x, ' y = ', y,' z = ', z);
end.
```

Observe que se a expressão lógica do primeiro comando *if then else* for verdadeira, o sistema executará outro comando *if then else*. Neste caso, dizemos que os comandos estão *aninhados*.

Observe também que escrevemos no programa algumas frases explicativas das ações pretendidas. Esta frases são chamadas *comentários* e devem ser escritas entre pares de caracteres {e}. Quando o compilador encontra o caractere *{* (abre chaves), procura um caractere *}* (fecha chaves) e desconsidera tudo o que vem entre os dois caracteres. Isto permite que o programador deixe registrado no próprio programa os comentários que ele achar conveniente. A prática de se colocar comentários nos programas é muito importante e facilita a sua compreensão. Como os programas discutidos neste livro serão precedidos de explicações prévias, a utilização de comentários aqui vai se restringir à indicação do objetivo do programa (como já vínhamos fazendo).

Cabe observar que a ação de armazenar o menor dos números na variável x pode ser feita de uma forma de compreensão mais simples (após dez anos de apresentação deste algoritmo para minhas turmas, um aluno me apresentou esta idéia). Basta colocar em x o menor dos conteúdos de x e de y e repetir este raciocínio com os conteúdos de x (talvez o novo conteúdo) e de z:

```
if (x > y)
    then
        begin
            Aux := x;
            x := y;
            y := Aux;
```

```
            end;
    if (x > z)
        then
            begin
                Aux := x;
                x := z;
                z := Aux;
            end;
```

7. Um outro exemplo que ilustra muito bem a utilização do comando *if then else* é um programa para determinar as raízes de uma equação do segundo grau. Sabemos da matemática que uma equação $ax^2 + bx + c = 0$ só tem raízes reais se $b^2 - 4ac \geq 0$. Além disto, para que ela seja do segundo grau, deve-se ter $a \neq 0$. Assim, um programa para encontrar as raízes reais (deixaremos o caso completo da determinação das raízes reais e complexas como exercício proposto) poderia ser o seguinte:

```
program EquacaoGrau2;
var a, b, c, x1, x2, Delta : real;
begin
    writeln('Digite os coeficientes');
    readln(a, b, c);
    if a <> 0
        then
            begin
                Delta := Sqr(b) - 4*a*c;
                if Delta >= 0
                    then
                        begin
                            x1 := (-b + SqrT(Delta))/(2*a);
                            x2 := (-b - SqrT(Delta))/(2*a);
                            writeln('Raízes da equação: ', x1, ' e ', x2);
                        end
                    else
                 writeln(' A equação dada não tem raízes reais')
     end
   else
        writeln('A equação não e do segundo grau');
end.
```

8. Imaginemos agora uma escola que adote no seu processo de avaliação a realização de quatro avaliações bimestrais e que o regime de aprovação dos alunos seja o seguinte:

i) Se a média das avaliações bimestrais for superior ou igual a 7,0, o aluno está *aprovado*, com média final igual à média das avaliações bimestrais.

ii) Se a média das avaliações bimestrais for inferior a 5,0, o aluno está *reprovado*, com média final igual à média das avaliações bimestrais.

iii) Não ocorrendo nenhum dos casos anteriores, o aluno se submete a uma *prova final* e a sua média final será a média ponderada desta prova final (com peso 4) e a média das avaliações bimestrais (com peso 6). Neste caso, o aluno estará aprovado se a sua média final for superior ou igual a 5,5.

O programa abaixo, recebendo as notas das avaliações bimestrais e, se for o caso, a nota da prova final, fornece a média final do aluno e a sua condição em relação à aprovação.

```
{Programa para verificar aprovação de um aluno}
program Avaliacao;
var Av1, Av2, Av3, Av4, MedBimestral, ProvaFinal, MedFinal : real;
begin
    writeln('Digite as notas das avaliações bimestrais');
    readln(Av1, Av2, Av3, Av4);
    MedBimestral := (Av1 + Av2 + Av3 + Av4)/4;
```

```
        MedFinal := MedBimestral;
        if (MedBimestral < 7) and (MedBimestral >= 5)
            then
                begin
                    writeln('Digite a nota da prova final');
                    readln(ProvaFinal);
                    MedFinal := (MedBimestral * 6 + ProvaFinal * 4)/10;
                end;
        if MedFinal >= 5.5
            then
                writeln('Aluno aprovado com media final igual a ', MedFinal)
            else
                writeln('Aluno reprovado com media final igual a ', MedFinal);
    end.
```

9. Para um exemplo de um programa que utiliza vários comandos *if then else* aninhados, suponhamos que uma empresa decidiu dar um aumento escalonado a seus funcionários de acordo com a seguinte regra: 13% para os salários inferiores ou iguais a R$ 200,00; 11% para os salários situados entre R$ 200,0 e R$ 400,00 (inclusive); 9 % para os salários entre R$ 400,00 e R$ 800,00 (inclusive) e 7% para os demais salários. Um programa que receba o salário atual de um funcionário e forneça o valor do seu novo salário poderia ser o seguinte.

```
    {Programa para atualizar salários}
    program AtualizaSalarios;
    var SalAtual, SalNovo, Aumento : real;
    begin
        writeln('Digite o valor do salário atual');
        readln(SalAtual);
        if SalAtual <= 200
            then
                Aumento := 1.13
            else
                if (SalAtual > 200) and (SalAtual <= 400)
                    then
                        Aumento := 1.11
                    else
                        if (SalAtual > 400) and (SalAtual <= 800)
                            then
                                Aumento := 1.09
                            else
                                Aumento := 1.07;
        SalNovo := SalAtual * Aumento;
        writeln('O salário de ', SalAtual:0:2,' será reajustado para ', SalNovo:0:2);
    end.
```

10. Um outro exemplo que utiliza comandos de seleção aninhados e em que a escolha da expressão lógica que controlará o comando *if then else* é importante é um programa que determine o número de dias de um mês (um programa como este seria parte integrante de um programa que manipulasse datas). Como os meses de trinta dias são quatro e os de trinta e um dias são sete, usamos os primeiros para o controle do comando de seleção.

```
    {Programa que determina o número de dias de um mês dado}
    program NumeroDeDias;
    var Mes, a, NumDias : integer;
    begin
        writeln('Digite o mês');
        readln(Mes);
        if (Mes = 4) or (Mes = 6) or (Mes = 9) or (Mes = 11)
            then
```

```
                NumDias := 30
            else
                if Mes = 2
                    then
                        begin
                            writeln('Digite o ano');
                            readln(a);
                            if (a mod 4 <> 0) or ((a mod 100 = 0) and (a mod 400 <> 0))
                                then
                                    NumDias := 28
                                else
                                    NumDias := 29;
                        end
                    else
                        NumDias := 31;
    writeln('Numero de dias do mes dado: ', NumDias);
end.
```

3.6 Sobre o ponto-e-vírgula e a endentação

O leitor deve ter observado que os programas em Pascal contêm muitos ponto-e-vírgulas: no final da identificação do programa, no final da declaração de variáveis e em finais de comandos. O ponto-e-vírgula, na verdade, é obrigatoriamente utilizado para delimitar áreas de programas e separar comandos. Assim, ele é obrigatório entre dois comandos e facultativo entre um comando e o delimitador *end*. Para facilitar a automatização, mesmo os ponto-e-vírgulas facultativos estão sendo colocados.

É necessário alertar que, como a opção *else* é parte integrante do comando *if then else*, antes de um *else* não existe ponto-e-vírgula. Quando, após um comando, o compilador encontra um ponto-e-vírgula, ele "entende" que o comando acabou e "procura" um outro comando ou um delimitador *end*. Encontrando uma opção *else*, ele acusa erro de compilação, pois *else* não é um comando.

Mesmo considerando que os compiladores da Linguagem Pascal não consideram espaços em branco nem mudanças de linha, estamos procurando (salvo raras exceções por questões de diagramação) escrever cada instrução em uma linha e a seqüência vinculada à estrutura de decisão com uma tabulação diferente (uma "maior" tabulação) da tabulação em que estão postos o *then* e o *else*. Esta forma de se editar um programa, chamada *indentação*, deve ser praticada por todo programador, pois ela facilita sobremaneira a legibilidade dos programas. Se o programa que determina as raízes reais de uma equação do segundo grau fosse digitado da forma seguinte,

```
                program EquacaoGrau2; var a, b, c, x1, x2, Delta
                : real; begin
writeln('Digite os coeficientes'); readln(a, b, c);
        if a <> 0 then
begin  Delta := Sqr(b) - 4*a*c;
        if Delta >= 0 then
        begin
x1 := (-b + SqrT(Delta))/(2*a); x2 := (-b - SqrT(Delta))/(2*a); writeln('As raízes da equação dada são ', x1,
' e ', x2);
                                        end
                            else writeln(' A equação dada não tem raízes reais') end else
writeln('A equação não e do segundo grau'); end.
```

ele seria executado da mesma forma, porém sua legibilidade seria muito prejudicada. Portanto, caro leitor, *indente* seus programas.

3.7 O comando *case*

Muitos programas são desenvolvidos de modo que eles possam realizar, de forma independente, várias tarefas. Por exemplo, um programa que gerencie um caixa eletrônico de um banco deve oferecer ao usuário algumas opções em relação à ação que ele pretende realizar na sua conta: a emissão do saldo atual, a emissão de um extrato, a realização de um saque ou a realização de um depósito. É comum que um programa que permita a realização de várias tarefas inicie apresentando ao usuário um *menu de opções* com a indicação das diversas tarefas que o programa pode executar e a permissão de que o usuário escolha a tarefa pretendida. Por exemplo, no caso do gerenciamento de um caixa eletrônico, poderíamos ter um programa que apresentasse, no início da sua execução, através de comandos de saída com mensagens, uma tela do tipo:

```
1. Extrato
2. Saldo
3. Saque
4. Depósito
                    Digite sua opção>
```

Como são várias as opções disponíveis (cada uma delas com uma seqüência específica de comandos) e só uma das opções será a escolhida, é útil uma estrutura que decida entre várias seqüências de comandos qual vai ser executada. O comando *case* tem este objetivo e deve ser escrito com a sintaxe

```
case Expressao of
    Lista de valores 1 : seqüência de comandos 1;
    . . .
    Lista de valores n : seqüência de comandos n;
    else seqüência de comandos x
end;
```

onde *Expressão* é uma expressão cujo valor é do tipo *integer*, *boolean* ou *char* (tipo ordenado) e *Listas de valores* é uma lista de valores, separados por vírgulas, do tipo de dado do valor da *Expressão* ou uma lista de intervalos deste tipo de dado. Neste último caso, os intervalos são dados através dos seus extremos separados por ponto ponto (ver exemplo 3 da seção 3.8).

Quando o comando *case* é executado, a *Expressao* é avaliada e então a seqüência cuja lista de valores contém o seu valor é executada. Se o valor da expressão não se encontra em nenhuma das listas de valores, a seqüência *x* será executada. A opção *else* é facultativa: se ela não existir e o valor da *Expressao* não se encontrar em nenhuma das listas de valores, nada é executado e o processamento vai para a instrução que segue a estrutura *case*.

No caso do programa que gerencie um caixa eletrônico poderíamos ter um programa que contivesse as seguinte declarações e instruções:

```
program GerenciaCaixaEletronico;
var Opcao : char;
    . . .
begin
    writeln('1. Extrato');
    writeln('2. Saldo');
    writeln('3. Saque');
    writeln('4. Depósito');
    write('                    Digite sua opção>   ');
    readln(Opcao);
    case Opcao of
        '1' : seqüência de comandos para emissão de um extrato;
        '2' : seqüência de comandos para emissão de saldo;
        '3' : seqüência de comandos para realização de um saque;
        '4' : seqüência de comando para realização de um depósito;
        else
            writeln(Chr(7), 'Opcao invalida');
```

```
end;
. . .
```

Naturalmente, o programa deveria possuir algum "mecanismo" que retornasse ao menu de opções quando a opção escolhida não fosse válida. Isto será visto no capítulo 4. Outra observação importante é que os sistemas visuais atuais (Delphi, Visual Basic, VisualC++) permitem que (facilmente em termos de programação) estes menus sejam apresentados de maneira bem agradável aos olhos, com a utilização de *botões* e *banners*, de modo que a escolha da opção seja feita através do *mause*, ou mesmo através de *toque* na tela. Estes formatos de menus podem ser feitos em Pascal através da utilização de *funções* e *procedimentos* contidos na *unit Graph*, mas isto foge do escopo deste livro.

3.8 Exemplos Parte V

1. O exemplo 10 da seção 3.5, que determina o número de dias de um mês dado, ficaria bem mais simples com a utilização do comando *case*:

```
{Programa que determina o número de dias de um mês dado}
program NumeroDeDias;
var Mes, Ano, NumDias : integer;
begin
    writeln('Digite o mes');
    readln(Mes);
    case Mes of
        4, 6, 9, 11 : NumDias := 30;
        2 : begin
                writeln('Digite o ano');
                readln(Ano)
                if ((Ano mod 4 = 0) and (Ano mod 100 <> 0)) or (Ano mod 400 <> 0)
                    then
                        NumDias := 29
            else
                        NumDias := 28;
            end;
        else
            NumDias := 31;
    end;
    writeln('O Mes ', Mes,' tem ', NumDias, ' dias');
end.
```

Observe que se o mês de entrada for 2, o programa pede o ano para determinar se ele é bissexto. Observe também que se o mês digitado for 1, 3, 5, 7, 8, 10 ou 12, a opção *else* será executada. Evidentemente, fica faltando discutir a possibilidade de uma entrada inválida como, por exemplo, 13. Isto será discutido num dos capítulos seguintes.

2. Um outro exemplo interessante de utilização do comando *case* é um programa que determine o dia da semana de uma data dada. Tomando como base o ano de 1600 (em 1582 o Papa Gregório III instituiu mudanças no calendário então vigente) e sabendo que o dia primeiro deste ano foi um sábado, para se determinar o dia da semana de uma data dada, basta se calcular o número de dias decorridos entre a tal data e o dia 01/01/1600. Como a associação do dia da semana a uma data é periódica, de período 7, o resto da divisão do número de dias referido acima por 7 indica a relação entre o dia da semana procurado e o sábado: se o tal resto for 1, o dia da semana é sábado; se o resto for 2, o dia da semana é domingo, e assim sucessivamente.

Para se calcular o número de dias entre uma data dada e 01/01/1600, basta multiplicar o número de anos por 365, acrescentar a quantidade de anos bissextos e o número de dias decorridos no ano corrente.

```
program DiaDaSemana;
var Dia, Mes, Ano, DiasDoAno, Dias31, AnosBiss : integer;
    Anos, NumDias : longint;
    Biss : boolean;
```

```
begin
    writeln('Digite dia, mes e ano ');
    readln(Dia, Mes, Ano);
    Biss := true;
    {Verifica se o ano e bissexto}
    if ((Ano mod 4 <> 0) or ((Ano mod 100 = 0) and (Ano mod 400 <> 0)))
        then
            Biss := false;
    Anos := Ano - 1600;
    {Numero de meses com 31 dias ate o mes dado}
    if (Mes < 9)
        then
            Dias31 := Mes div 2
        else
            Dias31 := (Mes + 1) div 2;
    {Numero de dias do ano dado, considerando fevereiro com tendo 30 dias}
    DiasDoAno := 30*(Mes - 1) + Dia + Dias31;
    {Retifica o numero de dias de fevereiro}
    if (Mes > 2)
        then
            if Biss
                then
                    DiasDoAno := DiasDoAno – 1
                else
                    DiasDoAno := DiasDoAno - 2;
    {Numero de anos bissextos entre o ano dado (exclusive) e 1600}
    if Biss
        then
            AnosBiss := (Ano div 4 - 400) - (Ano div 100 - 16) + (Ano div 400 - 4)
        else
            AnosBiss := (Ano div 4 - 400) - (Ano div 100 - 16) + (Ano div 400 - 4) + 1;
    {Numero de dias entre a data dada e 01/01/1600}
    NumDias := Anos*365 + DiasDoAno + AnosBiss;
    {Dia da semana}
    write('O dia ', Dia, '/', Mes,'/', Ano, ' caiu(caira) num(a) ');
    case NumDias mod 7 of
        1 : writeln(' Sabado');
        2 : writeln(' Domingo');
        3 : writeln(' Segunda');
        4 : writeln(' Terca');
        5 : writeln(' Quarta');
        6 : writeln(' Quinta');
        0 : writeln(' Sexta');
    end;
end.
```

Observe que a determinação do numero de anos bissextos entre 1600 e o ano da data dada foi feita de forma diferente daquela apresentada no exemplo 3 da seção 3.3.

3. Para exemplificar o comando *case* quando as listas de valores são listas de intervalos, imaginemos uma prova com 100 questões, cada uma valendo um ponto, devendo o resultado ser divulgado através de conceito de acordo com a seguinte tabela:

Intervalo da pontuação obtida	Conceito
0 a 49	D
50 a 69	C
70 a 89	B
90 a 100	A

Um programa que exibisse o conceito de uma dada pontuação obtida poderia ser o seguinte:

```
program ConvertePontuacaoEmConceito;
var NPontos: integer;
    Conc : char;
begin
    writeln('Digite a pontuação obtida');
    readln(NPontos);
    case NPontos of
        0 .. 49: Conc := 'D';
        50 .. 69 : Conc := 'C';
        70 .. 89 : Conc := 'B';
        90 .. 100: Conc := 'A';
    end;
    writeln('O numero de pontos igual a ', NPontos, 'corresponde a um conceito ', Conc);
end.
```

4. Para um outro exemplo de listas de intervalos e, complementarmente, para exemplificar a opção *else*, imagine que a empresa que vai atualizar os salários do exemplo 9 da seção 3.5 utilize apenas salários "inteiros". Lembramos que a tal empresa decidiu dar um aumento escalonado a seus funcionários de acordo com a seguinte regra: 13% para os salários inferiores ou iguais a R$ 200,00; 11% para os salários situados entre R$ 200,0 e R$ 400,00 (inclusive); 9 % para os salários entre R$ 400,00 e R$ 800,00 (inclusive) e 7% para os demais salários. Nestas condições, o exemplo referido poderia ser modificado para o seguinte programa:

```
{Programa para atualizar salários "inteiros"}
program AtualizaSalarios;
var SalarioAtual, SalarioNovo : integer;
    Aumento : real;
begin
    writeln('Digite o valor do salário atual');
    readln(SalarioAtual);
    case SalarioAtual of
        1 .. 200: Aumento := 1.13;
        201 .. 400 : Aumento := 1.11;
        401 .. 800 : Aumento := 1.09;
        else Aumento := 1.07;
    end;
    SalarioNovo := Round(SalarioAtual * Aumento);
    writeln('O salário de ', SalarioAtual,',00 será reajustado para ', SalarioNovo,',00');
end.
```

Observe a utilização da opção *else*: qualquer salário superior a R$ 800,00 não pertencerá a nenhuma das listas de intervalos e então a seqüência vinculada à opção *else* será executada. Observe também a necessidade de que a variável *SalarioAtual* seja do tipo *integer*: e de que a expressão que controla um comando *case* deve retornar um valor de um tipo ordenado e o tipo *real* não o é.

3.9 Exercícios propostos

1. Escreva um programa que realize arredondamentos de números utilizando a regra usual da matemática: se a parte fracionária for maior do que ou igual a 0,5, o número é arredondado para o inteiro imediatamente superior, caso contrário, é arredondado para o inteiro imediatamente inferior.

2. Escreva um programa para verificar se um inteiro dado é um quadrado perfeito exibindo, nos casos afirmativos, sua raiz quadrada.

3. Escreva um programa para determinar o maior de três números dados.

4. Escreva um programa para classificar um triângulo de lados de comprimentos dados em *escaleno* (os três lados de comprimentos diferentes), *isósceles* (dois lados de comprimentos iguais) ou *equilátero* (os três

lados de comprimentos iguais).

5. Escreva um programa para verificar se um triângulo de lados de comprimentos dados é *retângulo* exibindo, nos casos afirmativos, sua *hipotenusa* e seus *catetos*.

6. Escreva um programa para determinar as raízes reais ou complexas de uma equação do segundo grau, dados os seus coeficientes.

7. Escreva um programa para determinar a idade de uma pessoa, em anos meses e dias, dadas a data (dia, mês e ano) do seu nascimento e a data (dia, mês e ano) atual.

8. Escreva um programa que, recebendo as quatro notas bimestrais de um aluno da escola referida no exemplo 8 da seção 3.5, forneça a nota mínima que ele deve obter na prova final para que ele seja aprovado.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 4 Estruturas de Repetição

4.1 Para que servem estruturas de repetição

Um locutor brasileiro ao narrar um jogo de basquete ou de futebol americano nos Estados Unidos recebe a informação do placar eletrônico sobre a temperatura do estádio medida em graus fahrenheit. Naturalmente, ele deve fornecer aos telespectadores brasileiros a temperatura em graus Celsius. Para isto, o locutor, de posse de um computador, poderia utilizar o programa abaixo, que foi solicitado no primeiro item do segundo exercício da seção 2.12.

```
{Programa que converte uma temperatura dada em Graus fahrenheit para Graus Celsius}
program ConvTemp;
var Celsius, Fahrenheit : real;
begin
    writeln('Digite a temperatura em graus Fahrenheit');
    readln(Fahrenheit);
    Celsius := 5*(Fahrenheit - 32)/9;
    writeln(Fahrenheit, ' graus Fahrenheit correspondem a ', Celsius, ' graus Celsius');
end.
```

Se o placar eletrônico indicasse uma temperatura de 60º F, o narrador executaria o programa com a entrada 60 e receberia a saída

A temperatura de 60 graus fahrenheit corresponde a 15.55 graus Celsius

Evidentemente, seria mais prático a produção da transmissão do evento disponibilizar para o locutor uma tabela contendo as temperaturas possíveis em graus fahrenheit e as correspondentes em graus Celsius. A confecção desta tabela poderia ser feita através de um programa que contivesse vários comandos que calculassem para cada temperatura em graus fahrenheit pretendida a correspondente temperatura em graus Celsius e exibissem estas temperaturas. Neste caso, não haveria necessidade de comando de entrada; porém, para cada temperatura em graus fahrenheit pretendida, haveria, pelo menos, um comando de atribuição e um comando de saída. Se a faixa de temperatura em graus fahrenheit a ser coberta pela tabela fosse de vinte graus a oitenta graus, teríamos um programa como o programa abaixo.

```
{Programa (muito ruim) que gera uma tabela de conversão de temperaturas em graus Fahrenheit para
graus Celsius}
program TabelaDeConversaoDeTemperatura
var fahrenheit : integer;
begin
    writeln(Tabela de conversão graus Fahrenheit/graus Celsius');
    writeln('-------------------------------------------------');
    writeln('    Fahrenheit        |            Celsius');
    writeln('-------------------------------------------------');
    Fahrenheit = 10;
    writeln('    ',Fahrenheit, '    |           ', 5.0*(Fahrenheit - 32)/9);
    Fahrenheit = 11;
    writeln('    ',Fahrenheit, '    |           ', 5.0*(Fahrenheit - 32)/9);
    . . .
    {Mais "uma porção" de comandos!}
    Fahrenheit = 80;
    writeln('    ',Fahrenheit, '    |           ', 5.0*(fahrenheit - 32)/9);
end.
```

Isto seria contornado se pudéssemos repetir a execução dos comandos que gerariam as temperaturas em graus fahrenheit e as correspondentes em graus Celsius. A linguagem Pascal possui os comandos *for*; *while do* e *repeat until*, chamados *estruturas de repetição* ou *laços*, cujas execuções redundam em repetições da execução de uma determinada seqüência de comandos.

4.2 O comando *for*

O comando *for* é uma estrutura de repetição que repete a execução de uma dada seqüência de comandos um número de vezes que pode ser determinado pelo próprio programa, e devendo ser escrito com a seguinte sintaxe:

```
for Variavel := Expressao1  to Expressao2 do
     seqüência de comandos
```

onde *Varialvel* é uma variável de um tipo ordenado (*integer*, *char* e *boolean*, para lembrar) que é chamada *variável de controle* da estrutura e *Expressao1* e *Expressao2* são expressões cujos valores são do tipo de dado da variável de controle (numa boa parte das vezes *Expressao1* e *Expressao2* são constantes, quando são chamadas *Valor inicial* e *Valor final*). Além disto, e como nas opções de um comando *if*, há a necessidade dos delimitadores *begin* e *end* quando a *seqüência de comandos* tem mais de um comando.

A semântica do comando *for* é a seguinte: quando da sua execução, o valor da *Expressao1* é armazenado na variável de controle e o conteúdo desta (*Vc*) é comparado com o valor da *Expressao2* (*VFinal*); se *Vc* <= *VFinal*, a seqüência de comandos é executada. Em seguida o conteúdo da variável de controle é automaticamente incrementado de uma unidade e nova comparação é feita com *VFinal*. Isto tudo se repete até que o conteúdo da variável de controle seja maior que o valor da *Expressao2*.

Por exemplo, o programa

```
var i : integer;
begin
    for i := 1 to 10 do
        write(i, '     ');
end.
```

exibe na tela

```
1    2    3    4    5    6    7    8    9    10
```

enquanto que o programa

```
var i : integer;
begin
    for i := 10 to 1 do
        write(i, '   ');
end.
```

não exibe nada na tela, pois, quando da execução do comando *for*, a variável de controle i recebe o valor 10 e este já é maior que o *Valor final* 1(um), implicando a não execução da seqüência de comandos nem uma única vez.

Se, por alguma razão, se pretende que a variável de controle seja decrementada e, portanto, que o *Valor final* seja menor que o *Valor inicial*, deve-se substituir a preposição *to* pela "preposição" *downto*. Assim, o programa

```
var i : integer;
begin
    for i := 5 downto 1 do
        write(i, '        ');
end.
```

exibe na tela:

```
5    4    3    2    1
```

Com a estrutura aqui referida, a questão da geração de uma tabela de conversão de temperaturas em graus fahrenheit para graus Celsius seria simples.

{Programa para gerar uma tabela de conversão de temperaturas em graus Fahrenheit para graus Celsius}

```
program TabelaDeConversaoFarenheitCelsius;
uses Crt;
var Farenheit : integer;
    Celsius: real;
begin
    ClrScr;
    writeln('Tabela de conversao graus Farenheit/graus Celsius');
    writeln('-----------------------------------------------');
    writeln('     Farenheit          |           Celsius');
    writeln('-----------------------------------------------');
    for Farenheit := 20 to 80 do
        begin
            Celsius := 5.0*(Farenheit - 32)/9;
            writeln('      ', Farenheit,'      ', Celsius:0:2);
        end;
end.
```

Na execução do comando *for*, a variável *Fahrenheit* é inicializada com o valor 20, que é comparado com 80, a correspondente temperatura em graus Celsius é calculada e os dois valores são exibidos. Em seguida, o conteúdo de *Fahrenheit* é incrementado de uma unidade e tudo se repete até que *fahrenheit* atinja o valor 81. Desta forma, a execução deste programa gera a seguinte tabela

Tabela de conversão graus fahrenheit/graus Celsius

fahrenheit	Celsius
20	-6.67
21	-5.11
22	-5.56
23	-5.00
.	.
.	.
.	.
79	26,11
80	26,67

Vale observar que, ao contrário de outras linguagens (C, por exemplo), a variável de controle tem que ser necessariamente de um tipo ordenado. Se quiséssemos que a tabela também fornecesse temperaturas em graus Fahrenheit fracionárias (meio em meio grau, por exemplo), teríamos de considerar uma outra variável do tipo *integer* para controlar o comando *for*:

```
program TabelaDeConversaoFarenheitCelsius;
uses Crt;
var i : integer;
    Celsius, Farenheit : real;
begin
    ClrScr;
    writeln('Tabela de conversao graus fahrenheit/graus Celsius');
    writeln('-----------------------------------------------');
    writeln('     Farenheit          |           Celsius');
    writeln('-----------------------------------------------');
    Farenheit := 20
    for i := 1 to 121 do
        begin
            Celsius := 5.0*(Farenheit - 32)/9;
            writeln('      ', Farenheit,'      ', Celsius:0:2);
```

```
            Farenheit := Farenheit + 0.5
        end;
    end.
```

Certamente com a estrutura de repetição a seguir, o programa anterior ficasse um pouco mais simples.

4.3 O comando *while*

Para introduzir uma nova estrutura de repetição e cotejá-la com o comando *for*, considere um programa para encontrar um *divisor próprio* de um inteiro dado (um *divisor próprio* de um inteiro *n* é um divisor de *n* menor que *n* e diferente de 1. Esta questão é importante na verificação da *primalidade* de um inteiro: um número que não tem divisores próprios é *primo*, conforme [Evaristo, J. 2002]). Com a utilização do comando *for*, teríamos a seguinte solução para esta questão.

```
{Programa que determina um divisor próprio de um inteiro}
program DivisorProprio;
var Num, i, Divisor : integer;
begin
    write('Digite um numero: ');
    readln(Num);
    Divisor := 0;
    for i := 2 to Num - 1 do
        if Num mod = 0
            then
                Divisor := i;
    if Divisor <> 0
        then
            writeln(Divisor, ' eh divisor proprio de ', Num);
        else
            writeln(Num, ' eh primo');
end.
```

Um problema com este programa é que ele retorna sempre, se existir, o maior divisor próprio. Isto significa que, se a entrada for um número par, a estrutura de repetição não é interrompida quando o divisor 2 é encontrado, o que, evidentemente, vai prejudicar a performance do programa. Este problema pode ser resolvido em alguns compiladores Pascal que permitem que uma variável de controle de um comando *for* tenha o seu conteúdo alterado dentro do próprio comando. Com isto, o programa acima ficaria da seguinte forma.

```
{Programa que determina um divisor próprio de um inteiro}
program DivisorProprio;
var Num, i, Divisor : integer;
begin
    write('Digite um numero: ');
    readln(Num);
    Divisor := 0;
    for i := 2 to Num - 1 do
        if Num mod = 0
            then
                begin
                    Divisor := i;
                    i := Num – 1;
                end;
    if Divisor <> 0
        then
            writeln(Divisor, ' eh divisor proprio de ', Num);
        else
            writeln(Num, ' eh primo');
```

```
end.
```

Nesta versão, quando o primeiro divisor próprio é encontrado, o comando i := Num – 1 e o seguinte incremento da variável *i* faz com que a execução do comando *for* seja interrompida. A prática de alterar o conteúdo da variável de controle não será aqui incentivada pelo fato de que outras linguagens não a permitem. Na verdade, a questão central é que o comando *for* deve ser utilizado quando o número de repetições de execução de uma seqüência de comandos é conhecido *a priori*. Quando isto não acontece (que é o caso do exemplo anterior: não se sabe *a priori* se e quando um divisor próprio vai ser encontrado), deve-se usar o comando *while*, que possui a seguinte sintaxe:

```
while Expressão lógica do
        seqüência de comandos
```

Quando da sua execução, a *Expressão lógica* é avaliada. Se for verdadeira, a seqüência de comandos é executada e o processamento retorna ao próprio comando *while*. Se for falsa, a seqüência não é executada e o processamento se transfere para o comando seguinte.

Naturalmente, pode ocorrer que a seqüência não seja executada nenhuma vez, isto acontecendo se a *Expressão lógica* for falsa quando da "primeira" execução do comando. Por outro lado, se a tal expressão lógica permanecer verdadeira, a seqüência de comandos terá sua execução infinitamente repetida o que implicará a não-execução da tarefa. Se isto acontece, dizemos que o programa está em *looping* e é necessária a digitação da combinação das teclas <Ctrl> + <Break> para interromper a sua execução.

Com o comando *while* as questões levantadas acima sobre o programa para determinar um divisor próprio de um inteiro dado são resolvidas e temos o seguinte programa:

```
{Programa que determina o menor divisor próprio de um inteiro}
program DivisorProprio;
var Num, Divisor : integer;
begin
    write('Digite um numero: ');
    readln(Num);
    Divisor := 2;
    while Num mod Divisor <> 0 do
        Divisor := Divisor + 1;
    if Divisor < Num
        then
            writeln(Divisor, ' eh divisor proprio de ', Num);
        else
            writeln(Num, ' e primo');
end.
```

Como todo inteiro é divisor de si mesmo, a estrutura *while* sempre será interrompida. Se foi interrompida com *Divisor* < *Num*, é porque um divisor próprio foi encontrado; se foi interrompida com *Divisor* = *Num*, então *Num* é primo.

Evidentemente esta versão é bem mais eficiente do que aquela que utilizava o comando *for* e o interessante é que ela ainda pode ser melhorada, pois a matemática prova que, se um inteiro não possui um divisor próprio menor do que sua raiz quadrada, então ele é primo (ver [Evaristo, J 2002]). Levando em conta este fato, teríamos o seguinte programa:

```
{Programa que determina o menor divisor próprio de um inteiro}
program DivisorProprio;
var Num, Divisor : integer;
begin
    write('Digite um numero: ');
    readln(Num);
    Divisor := 2;
    while (Num mod Divisor <> 0) and (Divisor <= SqrT(Num)) do
        Divisor := Divisor + 1;
    if Divisor <= SqrT(Num)
        then
```

```
            writeln(Divisor, ' e divisor proprio de ', Num);
        else
            writeln(Num, ' e primo');
end.
```

Na procura de se aprender a escrever programas mais eficientes, observe que a função *SqrT* foi executada duas vezes para o mesmo valor. Num caso como este, é mais interessante considerar uma variável para armazenar o valor da função e utilizar o conteúdo da variável nos outros comandos. Desta forma o programa anterior poderia ser assim escrito:

```
{Programa que determina o menor divisor próprio de um inteiro}
program DivisorProprio;
var Num, Divisor : integer;
    Raiz : real;
begin
    write('Digite um numero: ');
    readln(Num);
    Raiz := SqrT(Num);
    Divisor := 2;
    while (Num mod Divisor <> 0) and (Divisor <= Raiz) do
        Divisor := Divisor + 1;
    if Divisor <= Raiz
        then
            writeln(Divisor, ' e divisor proprio de ', Num);
        else
            writeln(Num, ' e primo');
end.
```

Aproveitando o ensejo, vale observar que o comando *Divisor* := 2; dos programas acima atribuiu um *valor inicial* à variável *Divisor*. Este valor era modificado quando não era um divisor de *Num*. Um comando de atribuição de um valor inicial a uma variável é chamado *inicialização da variável*.

Uma outra aplicação importante do comando *while* diz respeito a execuções sucessivas de um programa. O leitor deve ter observado que os programas anteriores são executados apenas para uma entrada. Se quisermos a sua execução para outra entrada, precisamos executar o programa de novo.

Pode-se repetir a execução de um programa quantas vezes se queira, colocando-o numa estrutura definida por um comando *while*, controlada pelo valor de algum dado de entrada. Neste caso, o valor que encerra a execução pode ser informado dentro da mensagem que indica a necessidade da digitação da entrada. O programa anterior poderia ser então escrito da seguinte forma.

```
{Programa que determina o menor divisor próprio de vários inteiros}
program DivisorProprio;
var Num, Divisor : integer;
    Raiz : real;
begin
    Num := 1;
    while Num <> 0 do
        begin
            write('Digite um numero (0 para encerrar): ');
            readln(Num);
            Raiz := SqrT(Num);
            Divisor := 2;
            while (Num mod Divisor <> 0) and (Divisor <= Raiz) do
                Divisor := Divisor + 1;
            if Divisor <= Raiz
                then
                    writeln(Divisor, ' eh divisor proprio de ', Num);
                else
                    writeln(Num, ' eh primo');
```

```
        end;
    end.
```

4.4 O comando *repeat until*

Como dissemos na seção anterior, o número de execuções da seqüência de comandos associada a um comando *while* pode ser zero. Há situações em que é importante se garantir a execução de uma seqüência de comandos pelo menos uma vez. Uma delas é a verificação da *consistência dos dados de entrada*. Esta ação consiste em se dotar o programa de recursos para recusar dados incompatíveis com a entrada do programa, só "recebendo" dados que satisfaçam às especificações (lógicas ou estabelecidas) dos dados de entrada. Por exemplo, se a entrada é um número correspondente a um mês do ano, o programa não deve aceitar uma entrada que seja menor do que 1 nem maior do que 12. Uma solução para esta questão utilizando o comando *while* poderia ser a seguinte:

```
. . .
var Mes : integer;
begin
    writeln('Digite o mes: ');
    readln(Mes);
    while (Mes < 1) or (Mes > 12)
        begin
            writeln(Chr(7), 'Digitacao errada! Digite de novo');
            writeln('Digite o mes: ');
            readln(Mes);
        end;
    . . .
```

Observe que, como a verificação da condição de repetição é feita no "início" do comando, há a necessidade de um comando *readln* antes da estrutura e outra dentro dela (para lembrar, a função *Chr(x)* retorna o caractere de código ASCII *x*; o caractere de código ASCII igual a 7 é um caractere *não imprimível* e a execução de *writeln(Chr(7))* faz com que o sistema emita um sinal sonoro, um *beep*).

O comando *repeat until* define uma estrutura de repetição que garante que uma seqüência de comandos seja executada pelo menos uma vez. Sua sintaxe é

```
repeat
    seqüência de comandos;
until Expressão lógica;
```

e sua semântica é a seguinte: a seqüência de comandos é executada e a *Expressão* é avaliada; se o valor da *Expressão* for *false*, a seqüência de comandos é novamente executada e tudo se repete; do contrário, o comando que segue a estrutura é executado. Isto significa que a execução da seqüência de comandos será repetida até que a *Expressão lógica* seja verdadeira.

A consistência da entrada de um dado relativo a um mês utilizando um comando *repeat* poderia ser a seguinte.

```
. . .
var Mes : integer;
begin
    repeat
        writeln('Digite o mes: ');
        readln(Mes);
        if  (Mes < 1) or (Mes > 12)
            then
                writeln(Chr(7), 'Digitacao errada! Digite de novo');
    until (Mes >= 1) and (Mes <= 12);
. . .
```

Pode-se também utilizar o comando *repeat until* para execuções sucessivas de um programa. Neste caso,

é comum se fazer uma pergunta do tipo *Deseja continuar (S/N)?* após cada execução. Naturalmente, é necessária uma variável do tipo char que receba a resposta do usuário e que será utilizada para controlar a estrutura de repetição. Teríamos algo como:

```
var Resp : char;
. . .
repeat
     {seqüência de comandos do programa propriamente dito}
 writeln('Deseja continuar (S/N)?');
 readln(Resp);
until UpCase(Resp) = 'N';
```

Vale lembrar que a função *UpCase* retorna o argumento no formato maiúsculo. Esta função foi ativada aqui para que o usuário não se preocupe em digitar como resposta letras maiúsculas. Qualquer letra que for digitada, a função a torna maiúscula e o sistema a compara com S (maiúsculo).

4.5 Exemplos Parte VI

1. Consideremos um programa para determinar a soma dos *n* primeiros números pares positivos, *n* dado. Por exemplo, se for fornecido para *n* o valor 6, o programa deve retornar 42, pois 2 + 4 + 6 + 8 + 10 + 12 = 42. Naturalmente, o sistema pode gerar os números pares que se pretende somar, através do comando Par := 2 e da repetição do comando Par = Par + 2. Naturalmente, também, para que o sistema gere o próximo par, o anterior já deverá ter sido somado. Isto pode ser feito através do comando Soma = 0 e da repetição do comando Soma = Soma + Par. Temos então o seguinte programa.

```
{Programa que soma os n primeiros números pares, n dado}
program SomaPares;
var Soma, Par, n, i : integer;
begin
    write('Digite o valor de n: ');
    readln(n);
    Par := 2;
    Soma := 0;
    for i := 1 to n do
        begin
            Soma := Soma + Par;
            Par := Par + 2;
        end;
    writeln('Soma dos ', n, ' primeiros números pares:  ', Soma);
end.
```

Observe que os comandos Par = 2 e Soma = 0 atribuem um valor inicial às variáveis para que estes valores iniciais possam ser utilizados nas primeiras execuções dos comandos Soma = Soma + Impar e Impar = Impar + 2. Como já dissemos, chamamos um comando que atribui valor inicial a uma variável para que este valor possa ser utilizado na primeira execução de um comando que terá sua execução repetida denominada *inicialização da variável.*

Uma outra observação interessante é que, como existe uma fórmula que dá o i-ésimo número par ($a_i = 2i$), o programa acima poderia ser escrito de uma forma mais elegante, prescindindo, inclusive, da variável *Par*.

```
{Programa que soma os n primeiros números pares positivos, n dado}
 program SomaPares;
var Soma, n, i : integer;
begin
    writeln('Digite o valor de n: ');
    readln(n);
    Soma := 0;
    for i := 1 to n do
```

```
            Soma := Soma + 2*i;
        writeln('Soma dos ', n, ' primeiros números pares:  ', Soma);
    end.
```

Optamos por apresentar a primeira versão pelo fato de que nem sempre a fórmula para gerar os termos da seqüência que se pretende somar é tão simples ou é muito conhecida. Por exemplo, o exercício número 2 da seção 4.6 pede para somar os quadrados dos *n* primeiros números naturais e, neste caso, embora a fórmula exista, ela não é tão conhecida.

2. Um dos exemplos da seção anterior apresentava um programa que determinava, se existisse, um divisor próprio de um inteiro dado. Imaginemos agora que queiramos um programa que apresente a lista de todos os divisores próprios de um inteiro *n* dado. Neste caso, o programa pode percorrer todos os inteiros desde um até a metade de n verificando se cada um deles é um seu divisor. Temos então o seguinte programa.

```
{Programa que exibe os divisores próprios de um inteiro dado}
program Divisores;
var n, i : integer;
    Primo : boolean;
begin
    Primo := true;
    write('Digite o numero: ');
    readln(n);
    for i := 2 to (n div 2) do
        if n mod i = 0
            then
                begin
                    writeln(i);
                    Primo := false;
                end;
    if Primo
        then
            writeln(n, ' eh primo');
end.
```

Vale observar que, ao contrário do que foi dito na seção 2.9, os valores de saída deste programa não estão sendo armazenados. O que acontece é que ainda não temos condições de armazenar uma quantidade indefinida de elementos. Este problema será resolvido no capítulo 6. Vale observar também que a variável *Primo* é utilizada para detectar entradas correspondentes a números primos. Ela é inicializada com o valor *true* e seu conteúdo é modificado para *false* quando um divisor é encontrado.

3. Na seção 1.7 discutimos um algoritmo que determinava o quociente e o resto da divisão entre dois inteiros positivos dados. Embora os compiladores da linguagem Pascal possuam os operadores *mod* e *div* que calculam o resto e o quociente de uma divisão inteira entre dois inteiros positivos, vamos apresentar a implementação do algoritmo referido. Esta apresentação se justifica pelo fato de que este é um excelente exemplo de lógica de programação e também pelo fato de que o leitor pode um dia utilizar um sistema que não possua tais operadores.

```
{Programa que determina o quociente e o resto da divisao entre dois inteiros positivos}
var Divid, Divis, Quoc, Rest : integer;
begin
    writeln('Digite o dividendo e o divisor (diferente de zero!) ');
    readln(Divid,  Divis);
    Quoc := 1;
    while Quoc * Divis <= Divid do
        Quoc := Quoc + 1;
    Quoc := Quoc - 1;
    Rest := Divid - Quoc * Divis;
    writeln('Quociente e resto da divisao ', Divid, ' por ', Divis, ': ', Quoc, Rest);
end.
```

4. Em muitos casos, há necessidade de que um dos comandos da seqüência que terá sua execução repetida através de uma estrutura de repetição seja uma outra estrutura de repetição (num caso deste dizemos que as estruturas estão *aninhadas*). Para um exemplo, sejam A = {1, 2, 3, ..., n} e um programa que pretenda exibir o *produto cartesiano* AxA. Observe que, para isto, para cada valor da primeira componente o programa deve gerar todas as segundas componentes. Devemos ter, portanto, uma estrutura de repetição para gerar as primeiras componentes e uma outra, vinculada a cada valor da primeira componente, para gerar as segundas.

```
{Programa para gerar um produto cartesiano}
var n, i, j : integer;
begin
    write('Digite o numero de elementos do conjunto: ');
    readln(n);
    write('{');
    for i := 1 to n do
        for j := 1 to n do
            if (i < n) or (j < n)
                then
                    write('(', i, ', ', j, '), ')
                else
                    write('(', i, ', ', j, ')');
    writeln('}');
end.
```

Observe que o comando *if* se justifica para que não haja uma vírgula após o último par exibido.

5. É interessante observar que a variável de controle da estrutura *interna* pode depender da variável de controle da estrutura *externa*. Por exemplo, se em vez dos pares ordenados, quiséssemos os subconjuntos do conjunto A com dois elementos, o programa não deveria exibir o subconjunto, por exemplo, {1, 1}, que possui um só elemento, e deveria exibir apenas um dos subconjuntos {1, 2} e {2, 1}, já que eles são iguais. Isto pode ser obtido inicializando *j* com uma unidade maior do que o valor de *i*.

```
{Programa para gerar um conjunto de subconjuntos de um conjunto}
var n, i, j : integer;
begin
    write('Digite o numero de elementos do conjunto: '); readln(n);
    for i := 1 to n do
        for j := i + 1 to n do
           writeln('{',i, ', ', j,'}');
end.
```

6. Vejamos um programa para o cálculo da média de uma dada quantidade de números. Na seção 1.7, discutimos um algoritmo para determinar a média de 10 000 números dados. Na ocasião, discutimos que utilizaríamos uma única variável para receber os números, sendo que um valor subseqüente só seria solicitado depois que o anterior fosse "processado". A diferença agora é que a quantidade de números será um dado de entrada, o que torna o programa de aplicação mais variada. Como a quantidade de números será dada, pode-se utilizar uma estrutura *for* para receber e somar os números.

```
{Programa para calcular a media de n números, n dado}
program MediaN;
var n, i : integer;
    Num, Soma, Media : real;
begin
    Soma := 0;
    writeln('Digite o numero de elementos: ');
    readln(n);
    writeln(' Digite os elementos:');
    for i := 1 to n do
        begin
            readln(Num);
```

```
            Soma := Soma + Num;
        end;
    Media := Soma/n;
    writeln('Media = ', Media);
end.
```

7. O exemplo acima tem o inconveniente de que sua execução exige que se saiba anteriormente a quantidade de números. Naturalmente, isto não ocorre na maioria dos casos. Vejamos então um programa para determinar a média de uma relação de números dados, sem que se conheça previamente a quantidade deles. Neste caso, não devemos utilizar o comando *for*, pois não sabemos o número de repetições! Assim, o comando *while* deve ser utilizado. Porém, uma pergunta deve ser formulada: qual a expressão lógica que controlará a estrutura? A solução é "acrescentar" à relação um valor sabidamente diferente dos valores da relação e utilizar este valor para controlar a repetição. Este valor aqui referido é chamado *flag*. Como dito logo acima, deve-se ter certeza que o *flag* não consta da relação. Isto não é complicado, pois, ao se escrever um programa, se tem conhecimento de que valores o programa vai manipular e a escolha do *flag* fica facilitada. Por exemplo, se o programa vai manipular números positivos, pode-se usar -1 para o *flag*. Além do *flag*, o programa necessita de uma variável (no caso, *Cont* de contador) que determine a quantidade de números da relação, pois este valor será utilizado no cálculo da média.

```
{Programa para calcular a media de uma relacao de números}
program MediaX;
var Cont : integer;
    Num, Soma, Media : real;
begin
    Soma := 0;
    writeln(' Digite os elementos(-1 para encerrar):');
    readln(Num);
    Cont := 0;
    while Num <> -1 do
        begin
            Soma := Soma + Num;
            Cont := Cont + 1;
            readln(Num);
        end;
    Media := Soma/Cont;
    writeln('Media = ', Media);
end.
```

8. Na seção 1.7 apresentamos o *algoritmo de Euclides* para a determinação do *máximo divisor comum* de dois números dados. Para relembrar, vejamos como calcular o máximo divisor comum de 204 e 84.

	2	2	3
204	84	36	12
36	12	0	

O algoritmo é o seguinte: divide-se 204 por 84 obtendo-se resto 36; a partir daí, repetem-se divisões até que o resto seja zero, sendo o dividendo da divisão atual o divisor da divisão anterior e o divisor da divisão atual o resto da divisão anterior. O último divisor é o máximo divisor procurado.

Observe que a descrição deste algoritmo na linguagem coloquial não é muito simples. Ao contrário, escrever este algoritmo numa linguagem de programação é muito simples, pois uma estrutura de repetição e comandos de atribuição permitem que se obtenha facilmente a seqüência de divisões desejadas.

```
{Programa para determinar o maximo divisor comum de dois números positivos}
program MaxDivComum;
var x, y, a, b, Mdc, Resto : integer;
begin
    writeln('Digite os dois números  ');
    readln(x,  y);
    a := x;  b := y;
    Resto := a mod b;
```

```
    while Resto <> 0 do
        begin
            a := b;
            b := Resto;
            Resto := a mod b;
        end;
    Mdc := b;
    writeln('mdc(', x, ', ', y, ') =  ', Mdc);
end.
```

Note a necessidade da utilização das variáveis *a* e *b*. Elas são variáveis utilizadas no processamento e terão os seus conteúdos alterados durante a execução do programa. Se usássemos as variáveis x e y, os valores dos dados de entrada seriam perdidos, o que não deve ocorrer. No capítulo 5, quando estudarmos funções, estas variáveis terão outra conotação.

À primeira vista, o programa deveria inicialmente determinar o maior dos números *x* e *y*, armazenando-o em *a*. O quadro seguinte mostra que isto não é necessário, apresentando a simulação da execução do programa para x = 68 e y = 148.

x	y	a	b	Resto	Mdc
68	148	68	148	68	
		148	68	12	
		68	12	8	
		12	8	4	
		8	4	0	
					4

9. Um outro algoritmo matemático cuja implementação numa linguagem de programação apresenta um bom exemplo do uso de estruturas de repetição é o algoritmo para a determinação do *mínimo múltiplo comum* (mmc) de dois números dados. Como indica a própria denominação, o *mínimo múltiplo comum* de dois números é o menor número que é divisível pelos dois números. A matemática prova que o *mmc* de dois números é o produto dos divisores primos dos dois números, comuns ou não, ambos com as suas multiplicidades. Para que se obtenham os divisores primos, realizam-se divisões sucessivas pelos primos que são divisores de pelo menos um dos números.

A tabela seguinte mostra o cálculo do mínimo múltiplo comum dos números 360 e 420, como nos é ensinado no ensino fundamental.

360,	420	2
180,	210	2
90,	105	2
45,	105	3
15,	35	3
5,	35	5
1,	7	7
1,	1	MMC = 2*2*2*3*3*5*7 = 2 520

Observe que, quando um divisor primo é encontrado, repete-se a divisão, com o quociente no lugar do dividendo, até se obter um número que não seja múltiplo daquele divisor. Aí incrementa-se o tal divisor. Isto é feito até que ambos os quocientes sejam iguais a 1. Temos o seguinte programa:

```
{Programa para determinar o minimo multiplo comum de dois números positivos}
program MinMultComum;
var x, y, a, b, i, Mmc : integer;
begin
    writeln('Digite os dois números ');
    readln(x,  y);
    a := x;
    b := y;
    Mmc := 1;
    i := 2;
```

```
    while (a <> 1) or (b <> 1) do
        begin
            while (a mod i = 0) or (b mod i = 0) do
                begin
                    if a mod i = 0
                        then
                            a := a div i;
                            if  b mod i = 0
                                then
                                    b := b div i;
                                    Mmc := Mmc * i;
                                end;
                    i := i + 1;
                end;
    writeln('mmc(', x, ', ', y, ') =  ', Mmc);
end.
```

10. A questão do *mínimo múltiplo comum* é muito interessante como exemplo para a aprendizagem de programação pelo fato de que podemos apresentar um outro algoritmo de compreensão bem mais simples que o anterior.

A idéia é a seguinte: x + x, 2x + x, 3x + x, etc. são múltiplos de x. Para se obter o mínimo múltiplo comum basta que se tome o primeiro destes números que seja múltiplo também de y.

```
{Programa para determinar o minimo multiplo comum de dois números positivos}
program MinMultComum;
var x, y, Mmc : integer;
begin
    writeln('Digite os dois números ');
    readln( x,  y);
    Mmc := x;
    while Mmc mod  y <> 0 do
        Mmc := Mmc + x;
    writeln('mmc(', x,', ', y,') =  ', Mmc);
end.
```

4.6 Exercícios propostos

1. Mostre a configuração da tela após a execução do programa:

```
program ProgGeometrica;
var i, a, q, Termo : integer;
begin
    writeln('A configuração da tela apos a execução deste programa sera:')
    for i := 5 downto 1 do
        begin
            a = i;
            q = 3;
            Termo = a;
            write(6 – i,') ');
            while Termo <= 9 * a do
                begin
                    write(Termo, '  ');
                    Termo = Termo * q;
                end;
            writeln;
        end;
end.
```

2. Escreva um programa que determine a soma dos quadrados dos n primeiros números naturais, n dado.

3. Escreva um programa para calcular a soma dos n primeiros termos das seqüências abaixo, n dado.

a) $\left(\frac{1}{2}, \frac{3}{5}, \frac{5}{8}, ...\right)$

b) $\left(1, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, ...\right)$

4. O exemplo 10 da seção anterior apresentava uma solução para a questão do *mínimo múltiplo comum* de simples compreensão. Um problema que esta solução possui é que, se o primeiro valor digitado fosse muito menor do que o segundo, o número de repetições necessárias para se chegar ao *mmc* seria muito grande. Refaça o exemplo, tomando o maior dos números dados como base do raciocínio ali utilizado.

5. Um número inteiro é dito *perfeito* se o dobro dele é igual à soma de todos os seus divisores. Por exemplo, como os divisores de 6 são 1, 2, 3 e 6 e 1 + 2 + 3 + 6 = 12, 6 é perfeito. A matemática ainda não sabe se a quantidade de números perfeitos é ou não finita. Escreva um programa que liste todos os números perfeitos menores que um inteiro n dado.

6. O número 3 025 possui a seguinte característica: 30 + 25 = 55 e 55^2 = 3 025. Escreva um programa que escreva todos os números com quatro algarismos que possuem a citada característica.

7. Escreva um programa que escreva todos os pares de números de dois algarismos que apresentam a seguinte propriedade: o produto dos números não se altera se os dígitos são invertidos. Por exemplo, 93x13 = 39x31 = 1 209.

8. Escreva um programa para determinar o número de algarismos de um número inteiro positivo dado.

9. Escreva um programa que verifique se um dado número inteiro positivo é o produto de dois números primos. Por exemplo, 15 é o produto de dois primos, pois 15 = 3 x 5; 20 não é o produto de dois primos, pois 20 = 2 x 10 e 10 não é primo.

10. Quando um número não é produto de dois números primos, a matemática prova que ele pode ser escrito de maneira única como um produto de potências de números primos distintos. Este produto é chamado de *decomposição em fatores primos* do número e os expoentes são chamados de *multiplicidade* do primo respectivo. Por exemplo, 360 = 2^3x3^2x5. Escreva um programa que obtenha a decomposição em fatores primos de um inteiro dado.

11. Escreva um programa que transforme o computador numa *urna eletrônica* para eleição para presidente de um certo país, às quais concorrem os candidatos 83-Alibabá e 93-Alcapone. Cada voto deve ser dado pelo número do candidato, permitindo-se ainda o voto 00 para *voto em branco*. Qualquer voto diferente dos já citados é considerado *nulo*; em qualquer situação, o eleitor deve ser consultado quanto à confirmação do seu voto. No final da eleição, o programa deve emitir um relatório contendo a votação de cada candidato, a quantidade de votos em branco, a quantidade de votos nulos e o candidato eleito.

12. A *seqüência de Fibbonaci* é a seqüência (1, 1, 2, 3, 5, 8, 13, ...) definida por

$$a_n = \begin{cases} 1, se\ n = 1\ ou\ n = 2 \\ a_{n-1} + a_{n-2}, se\ n > 2 \end{cases}$$

Escreva um programa que determine o n-ésimo termo desta seqüência, n dado.

13. Os sistemas de computação que gerenciam caixas de lojas e supermercados fornecem ao operador, após a informação do valor do pagamento, o troco, em números decimais, que ele deve dar ao cliente. Talvez fosse interessante que, para otimizar a utilização das notas e das moedas de menor valor, visando a minimizar o problema da "falta de troco", o sistema fornecesse ao operador as quantidades de cada nota e de cada moeda para um "troco ótimo". Admitindo que o supermercado forneça também troco para pagamentos em cheque de qualquer valor, escreva um programa que, recebendo o valor da compra e o valor do pagamento, forneça o "troco ótimo" no sentido comentado acima.

14. A *série harmônica* $S = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} + ...$ é *divergente*. Isto significa que dado qualquer real *k*

existe n_0 tal que $1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n_0} > k$. Escreva um programa que dado um real *k* determine o menor inteiro n_0 tal que S > k. Por exemplo se k = 2, o programa deve fornecer n_0 = 4, pois $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = 2,083....$ e $1 + \frac{1}{2} + \frac{1}{3} = 1,8333....$

15. Escreva um programa que escreva todos os subconjuntos com três elementos do conjunto {1, 2, 3, ..., n}, *n* dado.

16. Dois números inteiros são ditos *amigos* se a soma dos divisores de cada um deles (menores que eles) é igual ao outro. Por exemplo, os divisores de 220 são 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 e 110 e 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 e os divisores de 284 são 1, 2, 4, 71 e 142 e 1 + 2 + 4 + 71 + 142 = 220. Escreva um programa que determine todos os pares de inteiros amigos menores um inteiro dado.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 5 Subprogramas

5.1 O que são subprogramas

O programa *Ordena3* do exemplo 6 da seção 3.5 continha algumas seqüências de comandos idênticas como

```
begin
    Aux := x;
    x := y;
    y := Aux;
end;
begin
    Aux := x;
    x := z;
    z := Aux;
end;
begin
    Aux := y;
    y := z;
    z := Aux;
end
```

cujos objetivos eram permutar os conteúdos das variáveis: *x* e *y*, na primeira seqüência; *x* e *z* na segunda e *y* e *z* na terceira seqüência. Ou seja, estas seqüências tinham o mesmo objetivo: permutar conteúdos de variáveis.

Para evitar repetições de seqüências idênticas (tanto na forma como no objetivo) de comandos, a maioria das linguagens de programação permite que se crie um "programa menor" dentro do programa que se está desenvolvendo de tal forma que o programa propriamente dito possa *ativar* a execução deste "programa menor" dentro de um dos seus comandos. Um "programa" contido em outro programa é chamado *subprograma* ou *sub-rotina* e pode conter declarações de variáveis, instruções, ativações de execuções de funções pré-definidas e ativações de outros subprogramas. Naturalmente, o objetivo de um subprograma deve ser a realização de alguma "subtarefa" específica da tarefa que o programa pretende realizar. Assim, pode-se escrever subprogramas para a leitura dos dados de entrada, para a saída do programa, para a determinação da média de vários elementos, para a troca dos conteúdos de uma variável (como no programa *Ordena3*), para o cálculo do máximo divisor comum de dois números dados etc. Normalmente, a realização da "subtarefa" para a qual a função foi escrita é chamada de *retorno* do subprograma, podendo um retorno ser a realização de uma ação genérica, como a leitura dos dados de entrada, ou um valor específico, como o cálculo do máximo divisor comum de dois números dados. Em Pascal, quando o retorno é a realização de uma ação genérica, o subprograma deve ser escrito como um *procedimento* e, quando o retorno é um valor específico, o subprograma deve ser escrito como uma *função*.

Em grande parte das vezes, a execução de um subprograma (*ativação* ou *chamada* do subprograma) requer "dados de entrada". Por exemplo, se fôssemos escrever o programa *Ordena3* utilizando subprogramas (e faremos isso), o subprograma que executaria a permuta dos conteúdos deveria conter algo que pudesse ser utilizado para indicar execuções diferentes em relações às variáveis cujos conteúdos se pretende permutar. Este algo são os *parâmetros* do subprograma e, como veremos a seguir, podem ser *variáveis* ou *referências a variáveis*. Para sua execução, um subprograma deve receber valores ou variáveis (em número igual ao número de parâmetros), sendo estes valores ou variáveis chamados de *argumentos*, que serão, na verdade, os *dados de entrada* do subprograma.

Além de permitir que repetições de seqüências de comandos idênticas sejam evitadas, a utilização de subprogramas permite que um programa seja escrito em *módulos*, o que possibilita as seguintes vantagens:

1. Melhor legibilidade do programa.
2. O programa pode ser desenvolvido em equipe, cada membro se encarregando do desenvolvimento

de alguns dos seus subprogramas.

3. A validação do programa através de testes pode ser realizada testando-se cada um dos subprogramas isoladamente, o que facilita a detecção e identificação de erros de lógica.

4. Atualizações posteriores do programa (atos chamados genericamente de *manutenção* do programa) são facilitadas, pelo fato de que apenas atualizações em alguns subprogramas são requeridas.

Outra aplicação importante de subprogramas se dá quando há necessidade de que o programa determine a mesma grandeza para valores diferentes. Um exemplo típico desta necessidade aparece num programa que determine medidas estatísticas (como *média aritmética*, *mediana*, *desvio médio*, *desvio padrão*) de uma relação de números. Como o *desvio médio* é a média aritmética dos valores absolutos dos desvios em relação à média, o seu cálculo exigirá a determinação da média aritmética da relação e a média aritmética dos desvios. Escreveremos então um subprograma para o cálculo da média de uma relação qualquer e o utilizaremos para os cálculos das duas médias necessárias. Este exemplo será visto no capítulo seguinte.

5.2 Procedimentos

Os procedimentos devem ser definidos antes do programa principal, como foi apresentado na seção 2.5, utilizando-se a seguinte sintaxe:

```
procedure Identificador(var lista de parâmetros : tipo de dado);
    declarações e definições
begin
    seqüência de comandos
end;
```

sendo as regras para o estabelecimento do identificador as mesmas utilizadas para a escolha do identificador do programa e a indicação *var* para os parâmetros opcional, o que será discutido adiante.

Como dissemos na seção anterior, os parâmetros de um procedimento servem para permitir que o mesmo seja "executado" para conjuntos de dados diferentes, os quais lhe serão fornecidos quando da sua ativação. O conjunto de parâmetros pode ser vazio, sendo que, neste caso, a ativação do procedimento executa sempre a mesma ação. Como também já foi dito, os dados (ou referências a variáveis, como veremos a seguir) que substituirão os parâmetros quando da execução do procedimento são chamados de *argumentos*. Naturalmente, os argumentos devem ser do mesmo tipo de dado parâmetro respectivo.

Do mesmo modo que num programa, num procedimento podem ser declaradas variáveis, definidos tipos de dados, relacionadas *units* e definidos outros subprogramas. Em qualquer caso, todo elemento declarado ou definido num procedimento só pode ser acessado pelos comandos deste procedimento. Usa-se comumente o termo *local* para se classificar um elemento que foi declarado ou definido num procedimento, utilizando-se o termo *global* para um elemento declarado ou definido no programa propriamente dito. Um elemento *global* (variável, tipo de dado, subprograma) pode ser acessado em qualquer parte do programa. No caso específico de variáveis, uma *variável global* existe durante toda a execução do programa enquanto que uma *variável local* só tem existência durante a ativação do procedimento.

5.3 Exemplos Parte VII

1. O programa Ordena3, já discutido anteriormente, escrito com a utilização de um procedimento, teria a seguinte forma:

```
program Ordena3;
var x, y, z : real;
    {procedimento que permuta os conteudos de duas variaveis}
    procedure Troca(var v1, v2 : real);
    var Aux : real;
    begin
        Aux := v1;
        v1 := v2;
        v2 := Aux;
```

```
    end;
  {programa principal}
  begin
     write('Digite os três números');
      readln(x, y, z);
      writeln('Valores dados: x = ', x:0:2, ', y = ', y:0:2, ' e z = ', z:0:2);
          if (x > y) or (x > z)
             then
                  if y > z
                     then
                          Troca(x, z)
                     else
                          Troca(x, y);
      if y > z
          then
              Troca(y, z);
     writeln('Valores dados, agora ordenados: x = ', x:0:2, ', y = ', y:0:2, 'e z = ', z:0:2);
  end.
```

Observe que a ativação de um procedimento é feita com a simples referência ao seu identificador junto com os argumentos com os quais se pretende que ele seja executado. Observe também que, através de um comentário, indicamos o objetivo do procedimento. Esta é uma prática que deve ser adotada por todo programador, pois facilita sobremaneira a leitura do programa.

2. Um programa com objetivos múltiplos deve oferecer uma tela com as tarefas que ele pode realizar para que o usuário escolha qual delas ele deseja. Uma tela com este objetivo é comumente chamada de *menu de opções* e funciona como uma *interface* entre o programa e o usuário.

Um menu de opções pode ser obtido através de um procedimento sem parâmetros. Por exemplo, um programa para gerenciar as contas correntes de uma banco pode conter o seguinte procedimento:

```
  procedure Menu;
  begin
      writeln('1 - Saldo');
      writeln('2 - Extrato');
      writeln('3 – Aplicação e resgate');
      writeln('4 - Depósitos');
      writeln('5 - Saques');
      write('                    Digite sua opção : ');
      readln(Opcao);
  end;
```

Naturalmente, o programa principal seria iniciado com a chamada deste procedimento, seguido de um comando *case* para, de acordo com a opção escolhida, executar a tarefa pretendida. Neste caso, a variável *Opção* deveria ser uma variável global.

Cabe relembrar o que foi dito na seção 3.7: com o surgimento das linguagens visuais (VisualBasic, Delphi e outras), atualmente os *menus de opções* são disponibilizados através de *interfaces gráficas* contendo *botões*, *banners* e outros elementos e as ativações dos subprogramas que executam a tarefa pretendida são feitas através de *mouses* ou mesmo através de toque manual na tela do computador. A *unit Graph* contém funções e procedimentos predefinidos que permitem que *interfaces gráficas* sejam criadas em Pascal, porém o estudo destas subrotinas não está no escopo deste livro.

3. Os comandos de entrada e de saída, *readln* e *writeln*, são, na verdade, procedimentos predefinidos do sistema. O comando *readln*, por exemplo, possui um conjunto de parâmetros, cujo número de elementos não é prefixado, podendo até ser um conjunto vazio, situação na qual necessita apenas da digitação da tecla <Enter> para sua execução. Por seu turno, o procedimento *writeln* pode receber como argumentos identificadores de variáveis, expressões, mensagens e até, como veremos no capítulo 10 (dez), arquivos. Do mesmo modo que o comando *readln*, o conjunto de parâmetros do comando *writeln* pode ser vazio, quando, neste caso, sua execução implica a mudança do cursor para a linha seguinte.

4. O Sistema Turbo Pascal oferece dois procedimentos para *incrementar* e *decrementar* uma variável. São os procedimentos *Inc* e *Dec* que podem ser ativados com um ou dois parâmetros. O primeiro (ou único) parâmetro é a variável que deve ser incrementada ou decrementada e o segundo parâmetro é o inteiro que dá o valor do incremento ou do decremento. Por exemplo, *Inc*(i, 2) incrementará de duas unidades o conteúdo da variável i; *Dec*(i) decrementará de uma unidade o conteúdo de i.

5.4 Funções

Como vimos acima, procedimentos são subprogramas que executam tarefas genéricas não necessariamente retornando algum valor. As *funções*, além de poderem executar ações genéricas, podem retornar valores, devendo ser declaradas de acordo com a seguinte sintaxe :

```
function Identificador(var lista de parâmetros : tipo de dado) : tipo de dado;
    declarações e definições
begin
    seqüência de comandos;
end;
```

Se a função deve retornar um valor, este valor será do tipo de dado fixado na sua declaração, que é necessariamente um *tipo de dado simples* ou uma *string* (o tipo de dado *string* será estudado no capítulo 7). Este tipo de dado associado à função é usualmente chamado de *tipo da função*. Neste caso, é necessário que na seqüência de comandos de uma função exista uma atribuição do tipo

```
Identificador := Expressão;
```

sendo justamente o valor desta *Expressão* o que será retornado para processamento, quando da ativação da função. Esta ativação deve ser feita num segundo membro de um comando de atribuição, como argumento de um comando de saída ou, de um modo geral, numa expressão.

Do mesmo modo que nos procedimentos, variáveis, tipos de dados e outros subprogramas podem ser definidos localmente.

5.5 Exemplos Parte VIII

1. O exercício 2, item c, da seção 2.12, solicitava um programa que realizasse a soma de duas frações ordinárias. O caro leitor deve ter desenvolvido a solução da questão e obtido um programa parecido com o seguinte:

```
program SomaFracoes;
var Num1, Den1, Num2, Den2, Num, Den: integer;
begin
    writeln('Digite as fracoes');
    readln(Num1, Den1, Num2, Den2);
    Num := Num1 * Den2 + Num2 * Den1;
    Den := Den1 * Den2;
    writeln('(', Num1, '/', Den1, ') + (', Num2, '/', Den2,') = (', Num, '/', Den, ')');
end.
```

Um problema com este programa é que ele fornece a soma das frações na forma de uma fração *redutível*, ao contrário do que os professores de matemática exigem (confesso, não sei o porquê desta exigência). Sabendo que, para *simplificar* uma fração, basta dividir seus termos pelo seu máximo divisor comum, este programa seria melhorado se acrescentássemos uma função que calculasse o *mdc* de dois números, utilizando-se este valor para simplificar a fração.

```
program SomaFracoes;
var Num1, Den1, Num2, Den2, Num, Den, Mdc: integer;
    {função que retorna o maximo divisor comum de dois inteiros dados}
    function MaxDivComum(x, y : integer) : integer;
    var Resto : integer;
    begin
```

```
            Resto := x mod y;
            while Resto <> 0 do
                begin
                    x := y;
                    y := Resto;
                    Resto := x mod y;
                end;
            MaxDivComum := y;
        end;
    {programa principal}
    begin
        writeln('Digite as fracoes');
        readln(Num1, Den1, Num2, Den2);
        Num := Num1 * Den2 + Num2 * Den1;
        Den := Den1 * Den2;
        Mdc := MaxDivComum(Num, Den);
        Num := Num div Mdc;
        Den := Den div Mdc;
        writeln('(', Num1, '/', Den1, ') + (', Num2, '/', Den2,') = (', Num, '/', Den, ')');
    end.
```

2. Sabendo que o *fatorial* de um inteiro não negativo *n* é o produto de todos os inteiros de 1 até *n*, sendo 1 se n = 1 ou n = 0, o exemplo a seguir apresenta uma função que retorna o valor do fatorial de um inteiro n, dado (n! = 1*2*3*...*n).

```
function Fatorial(m : integer) : longint;
var f : longint;
    i : integer;
begin
    f := 1;
    for i := 1 to m do
        f := f*i;
    Fatorial := f;
end;
```

O tipo da função foi definido como *longint*, porque o fatorial é uma função que assume valores relativamente grandes para argumentos pequenos. Por exemplo, Fat(8) = 40 320.

É necessário um certo cuidado com o identificador de uma função. À primeira vista poderia se pensar que a utilização da variável *f* no exemplo seria desnecessária, substituindo a seqüência de comandos por:

```
Fatorial := 1;
for i := 1 to m do
    Fatorial := Fatorial * i;
```

Ocorre que, como veremos na seção 5.7, Pascal oferece o recurso da *recursividade*, que nada mais é do que a possibilidade de que um subprograma ative a si próprio. Assim o sistema entenderia o *Fatorial* do segundo membro do comando

```
Fatorial := Fatorial * i
```

como uma nova ativação da função e aí iria reclamar a ausência dos argumentos.

3. O exemplo 2 da seção 4.5 apresentava um programa que detectava números primos. O exemplo a seguir apresenta uma função boleana que retorna *true* se o argumento for um inteiro primo.

```
function Primo(m : integer) : boolean;
var i : integer;
    Raiz : real;
begin
    i := 2;
    Raiz := SqrT(m);
```

```
    while (m mod i <> 0) and (i <= Raiz) do
        i := i + 1;
    if i <= Raiz
        then
            Primo := false
        else
            Primo := true;
end;
```

5.6 Passagem de parâmetros

No procedimento *Troca* do exemplo 1 da seção 5.3, as definições dos parâmetros eram precedidas da palavra reservada *var* o que não acontecia com as definições dos parâmetros da função *MaxDivComum* do exemplo 1 da seção 5.5. O que estes dois subprogramas possuem de diferentes é o seguinte: no primeiro, se pretendia que o procedimento tivesse influência nos conteúdos das variáveis passadas como argumentos enquanto que, no segundo, os conteúdos das variáveis passadas como argumentos não deveriam ser modificados pelos comandos da função.

Quando se pretende que comandos de um subprograma modifiquem conteúdos de variáveis globais, as declarações dos parâmetros respectivos devem ser precedidas da palavra reservada *var*. Se isto acontece, o argumento a ser passado para o parâmetro tem que ser uma variável e o subprograma pode alterar o seu conteúdo, pois o que é passado para o parâmetro é uma *referência* à variável argumento; tudo se passa como se o parâmetro recebesse a própria variável. Neste caso, dizemos que a passagem dos parâmetros é *por referência*. Quando não se pretende que comandos do subprograma não interfiram em conteúdos de variáveis globais, a declaração dos parâmetros não é precedida da palavra *var* e o parâmetro recebe um elemento do seu tipo de dado, isto podendo ser feito através de:

1. constantes do tipo de dado do parâmetro respectivo;
2. conteúdos de variáveis do mesmo tipo de dado;
3. expressões cujos resultados sejam daquele tipo.

Neste caso, a passagem de parâmetros é dita *por valor*. Resumindo: se a declaração de um parâmetro é precedida da palavra reservada *var*, o argumento tem que ser uma variável e comandos que alterem o parâmetro alterarão a variável; se a declaração do parâmetro não é precedida de *var*, o argumento é um valor (podendo ser passado como conteúdo de um variável) e os comandos do subprograma não interferem em conteúdos de variáveis globais.

Voltando a insistir: o procedimento *Troca* do programa *Ordena3*, apresentado na seção 5.3, tinha como objetivo permutar os conteúdos das variáveis globais utilizados como argumentos. Assim, as ações realizadas nos parâmetros deveriam se refletir nos conteúdos dos argumentos de ativação do procedimento. Isto implicou a necessidade de se utilizar a passagem por referência.

Um outro aspecto interessante é que a passagem de parâmetros por referência permite que se substitua uma função por um procedimento. Basta que se utilize um parâmetro por referência para retornar o valor para uma variável global. Por exemplo, um programa para calcular o fatorial de um inteiro dado, utilizando um procedimento, poderia ter a seguinte forma:

```
program Fatoriais;
var n : integer;
    Fat : longint;
   {Procedimento que retorna o fatorial de um inteiro}
    procedure Fatorial(m : integer; var f : longint);
    var i : integer;
    begin
        f := 1;
        for i := 1 to m  do
            f := f*i;
    end;
{Programa Principal}
```

```
begin
    write('Digite um inteiro');
    readln(n);
    Fatorial(n, Fat);
    writeln(n,'! = ', Fat);
end.
```

Na definição do procedimento *Fatorial*, se estabelece que o parâmetro *m* receberá um valor e que o parâmetro *f* receberá uma referência. Na ativação do procedimento, no programa principal, o conteúdo da variável global *n* é passado para o parâmetro *m* (passagem por valor) e a referência à variável *Fat* é passada para o parâmetro *f* (passagem por referência). Assim, todas as alterações no conteúdo do parâmetro *f* se refletirão no conteúdo da variável *Fat*.

A passagem de parâmetros *por referência* é muito importante quando se pretende que uma função retorne mais de um valor. Um destes valores pode ser retornado pelo comando

Identificador da função := Expressão;

e os demais podem ser retornados para variáveis que foram passadas *por referência* para parâmetros da função. Um exemplo disto ocorre na função *Confirma* que soluciona o exercício proposto 11 da seção 4.6 (agora com a utilização de funções). Para lembrar, queríamos um programa para transformar o computador numa urna eletrônica para a eleição, em segundo turno, para a presidência de um certo país, à qual concorrem dois candidatos: *Alibabá*, de número 83, e *Alcapone*, de número 93. A eleição permite ainda o voto em branco (número 22) e considera como voto nulo qualquer voto diferente dos anteriores. A função *Confirma* deve retornar dois valores: o primeiro para, no caso de confirmação do voto, permitir sua contabilização e o segundo para, ainda no caso de confirmação do voto, interromper a estrutura *repeat*, o que permitirá a recepção do voto seguinte. Observe também a passagem por referência do parâmetro da função *ComputaVoto*. Há necessidade de que seja desta forma, pelo fato de que esta função alterará conteúdos de variáveis diferentes.

```
{Programa que transforma um computador numa urna eletronica}
program UrnaEletronica;
uses Crt;
var Voto, Alibaba, Alcapone, Brancos, Nulos : integer;
     ConfVoto, Cont : char;
    {Funcao para confirmacao do voto'}
    function Confirma(s: string; var Conf : char): boolean;
    begin
        writeln('Voce votou em ', s, '! Confirma seu voto(S/N)');
        readln(Conf);
        if UpCase(Conf) = 'S'
            then
                Confirma := true
            else
                begin
                    writeln('Vote de novo');
                    Sound(400);
                    Delay(1000);
                    NoSound;
                    Confirma := false;
                end;
        end;
        {Funcao para computar um voto}
        function ComputaVoto(var v : integer) : integer;
        begin
            v := v + 1;
        end;
{Programa principal}
begin
    Clrscr;
```

```
Cont := 'S';
Alibaba := 0; Alcapone := 0; Brancos := 0; Nulos := 0;
while UpCase(Cont) = 'S' do
    begin
        repeat
            writeln('83 - Alibaba   93 - Alcapone   99 - Branco    Outro valor - Nulo');
            writeln('Digite seu voto');
            readln(Voto);
            case Voto of
                    83 : if Confirma('Alibaba', ConfVoto)
                            then
                                ComputaVoto(Alibaba);
                    93 : if Confirma('Alcapone', ConfVoto)
                            then
                                ComputaVoto(Alcapone);
                    00 : if Confirma('Branco', ConfVoto)
                            then
                                ComputaVoto(Brancos);
                    else
                        if Confirma('Nulo', ConfVoto)
                            then
                                ComputaVoto(Nulos);
            end;
            Clrscr;
        until ConfVoto = 's';
        writeln('Novo eleitor (S/N)?');
        readln(Cont);
    end;
Clrscr;
writeln('Resultado da eleicao');
writeln('          Alibaba: ', Alibaba);
writeln('          Alcapone: ', Alcapone);
writeln('          Brancos: ', Brancos);
writeln('          Nulos: ', Nulos);
writeln;
writeln;
write('Candidato eleito: ');
if Alibaba > Alcapone
    then
        writeln('Alibaba')
    else
        if Alibaba < Alcapone
            then
                writeln('Alcapone')
            else
                writeln('Eleicao empatada');
end.
```

O procedimento predefinido *Sound* possui um parâmetro *f* do tipo *word* e sua execução faz com que o sistema emita um som de freqüência *f* hertz. O som emitido pelo procedimento *Sound* só será interrompido quando da execução do procedimento *NoSound* que não tem parâmetros (estes procedimento podem ser utilizados para transformar o teclado num instrumento musical; que tal tentar?). O procedimento *Delay* tem um parâmetro *m* e faz com que o sistema interrompa a execução do programa por *m* milissegundos.

Como já foi dito, o tipo *string* do parâmetro *s* da função *Confirma* será estudado no capítulo 7. Para compreender o programa acima, basta saber que uma variável do tipo *string* pode armazenar nomes (de um modo geral, um *cadeia de caracteres*). Desta forma, podemos ativar a função *Confirma* com os argumentos '*Alibaba*', '*Alcapone*', '*Nulo*' e '*Branco*' para que seja possível dizer ao eleitor (através do comando *writeln*)

qual foi o seu voto e ele então possa confirmá-lo ou não.

5.7 Recursividade

Algumas funções matemáticas clássicas podem ser estabelecidas de tal forma que as suas definições utilizem, de modo recorrente, a própria função que se está definindo. Um exemplo trivial (no bom sentido) de um caso como este é a função *fatorial*. Como dissemos no exemplo 2 da seção 5.5, o fatorial de um número inteiro não-negativo *n* é o produto de todos os números naturais de 1 até o referido *n* ou seja n! = 1 . 2 . 3 . ... . n, sendo igual a 1 quando n = 0 ou n = 1. Como mostrou o referido exemplo, é muito simples se escrever uma função que calcule o fatorial de *n*:. Basta se inicializar uma variável com 1 e, numa estrutura de repetição, calcular os produtos 1 x 2 = 2, 2 x 3 = 6; 6 x 4 = 24; 24 x 5 = 120; ...; etc., até multiplicar todos os naturais até *n*.

Embora o conceito anterior seja de simples compreensão, pode-se obter uma definição mais elegante para o fatorial de um inteiro não-negativo *n*:

$$n! = \begin{cases} 1, se\ n = 0\ ou\ n = 1 \\ n\ .\ (n - 1)!, n > 1 \end{cases}$$

Desta forma, o fatorial de *n* é definido a partir dos fatoriais dos naturais menores que *n*. Isto significa que, para o cálculo do fatorial de um determinado número natural, há necessidade de que se *recorra* aos fatoriais dos naturais anteriores. Por exemplo,

4! = 4 . 3! = 4 . (3 . 2!) = (4 . 3) . (2 . 1!) = 4 . 3 . 2 . 1 = 24.

Uma definição com estas características é dita uma definição por *recorrência* ou uma definição *recursiva*.

Um outro exemplo de uma definição recursiva foi dada no exercício 12 da seção 4.6: a *seqüência de Fibbonaci* é a seqüência (a_n) definida por

$$a_n = \begin{cases} 1, se\ n = 1\ ou\ n = 2 \\ a_{n-1} + a_{n-2}, se\ n > 2 \end{cases}$$

Observe que o termo de ordem *n* é definido a partir de termos anteriores. Isto significa que, para o cálculo de um determinado termo, há necessidade de que se *recorra* a valores de todos os termos anteriores. Por exemplo, para a determinação de a_5 necessitamos conhecer a_4 e a_3; para a determinação destes dois, necessitamos conhecer a_2 e a_1.

Naturalmente, uma definição recursiva deve conter uma condição que interrompa a *recorrência*. Esta condição é chamada *condição de escape*. No caso do fatorial, a condição de escape é n = 0 ou n = 1; na seqüência de Fibbonaci, a condição de escape é n = 1 ou n = 2. A expressão que realiza propriamente a recorrência pode ser chamada *expressão de recorrência*.

O que surpreende a todos que começam a aprendizagem de programação é que, de um modo geral, as linguagens de programação oferecem recursos para implementação de funções recursivas da mesma maneira que elas são escritas em matemática. Por exemplo, a implementação recursiva do fatorial pode ser feita em Pascal simplesmente da seguinte forma:

```
{Programa para gerar uma tabela de fatoriais}
program TabelaFatoriais;
var i : integer;
    {Funcao recursiva para o cálculo do fatorial de um inteiro não negativo}
    Function FatRec (n : integer) : longint;
    begin
        if (n = 0) or (n = 1)
            then
                FatRec := 1
            else
                FatRec := (n * FatRec(n - 1));
```

```
    end;
  {Programa principal}
  begin
    for i := 1 to 10 do
      writeln(i, '! = ', FatRec(i));
  end.
```

É interessante ter uma idéia do que acontece na recursividade. Quando se ativa uma função recursiva, cada nova chamada da mesma é empilhada na chamada *pilha de recursão* até que a condição de escape seja atingida. A partir daí, cada ativação pendente é desempilhada (evidentemente, na ordem inversa do empilhamento) e as operações vão sendo realizadas. No programa acima, quando i = 5, temos a seguinte seqüência de operações:

1. Após a ativação de FatRec(5)

FatRec(5)	n	
5*FatRec(4)	5	

2. Após a ativação de FatRec(4)

FatRec(5)	n	FatRec(4)	n	
5*FatRec(4)	5	4*FatRec(3)	3	

3. Após a ativação de FatRec(3)

FatRec(5)	n	FatRec(4)	n	FatRec(3)	n	
5*FatRec(4)	5	4*FatRec(3)	3	3*FatRec(2)	2	

4. Após a ativação de FatRec(2)

FatRec(5)	n	FatRec(4)	n	FatRec(3)	n	FatRec(2)	n	
5*FatRec(4)	5	4*FatRec(3)	3	3*FatRec(2)	2	2*FatRec(1)	1	

5. Após a ativação de FatRec(1)

FatRec(5)	n	FatRec(4)	n	FatRec(3)	n	FatRec(2)	n	
5*FatRec(4)	5	4*FatRec(3)	3	3*FatRec(2)	2	2*1 = 2	1	

FatRec(5)	n	FatRec(4)	n	FatRec(3)	n	
5*FatRec(4)	5	4*FatRec(3)	3	3*2 = 6	2	

FatRec(5)	n	FatRec(4)	n	
5*FatRec(4)	5	4*6 = 24	3	

FatRec(5)	n	
5*24 = 120	5	

Embora a utilização da recursividade apresente a vantagem de programas mais simples, ela traz o inconveniente de sacrificar a eficiência do programa. Isto ocorre devido à necessidade de chamadas sucessivas da função e das operações de empilhamento e desempilhamento, o que demanda um tempo maior de computação e uma maior necessidade de uso de memória. Esta observação faz com que a solução não-recursiva (chamada *solução iterativa*) seja preferível. No capítulo 10, apresentaremos um exemplo de uma função recursiva que é tão eficiente quanto a função iterativa.

Um outro exemplo interessante de recursividade é a implementação do jogo conhecido como *Torre de Hanói* que foi objeto do exercício proposto 2 da seção 1.12. Para lembrar, este jogo consiste de três torres chamadas *origem*, *destino* e *auxiliar* e um conjunto de *n* discos de diâmetros diferentes, colocados na torre *origem*, na ordem decrescente dos seus diâmetros. O objetivo do jogo é, movendo um único disco de cada vez e não podendo colocar um disco sobre outro de diâmetro menor, transportar todos os discos para a torre *destino*, podendo usar a torre *auxiliar* como passagem intermediária dos discos.

Indicando com torre 1 → torre 2 o movimento do disco que no momento está na parte superior da torre 1 para a torre 2, teríamos a seguinte solução para o caso n = 2:

1. origem → auxiliar
2. origem → destino
3. auxiliar → destino

Para n = 3, a solução seria:

1. origem → destino
2. origem → auxiliar
3. destino → auxiliar
4. origem → destino
5. auxiliar → origem
6. auxiliar → destino
7. origem → destino

Observe que os três movimentos iniciais transferem dois discos da torre *origem* para a torre *auxiliar*, utilizando a torre *destino* como auxiliar; o quarto movimento transfere o maior dos discos da *origem* para *destino* e os últimos movimentos transferem os dois discos que estão na *auxiliar* para *destino*, utilizando *origem* como torre auxiliar.

Assim, a operação Move(3, origem, auxiliar, destino) - move três discos da *origem* para *destino* usando *auxiliar* como torre auxiliar - pode ser decomposta em três etapas:

1. Move(2, origem, destino, auxiliar) - move dois discos de origem para auxiliar usando destino como auxiliar;
2. Move um disco de origem para destino
3. Move(2, auxiliar, origem, destino) - move dois discos de auxiliar para destino usando origem como auxiliar.

O interessante é que é fácil mostrar que este raciocínio se generaliza para *n* discos, de modo que a operação Move(n, a, b, c) pode ser obtida com as seguintes operações:

1. Move(n-1, a, c, b)
2. Move um disco de a para c
3. Move(n-1, b, a, c)

O mais interessante ainda é que isto pode ser implementado em Pascal, através do seguinte programa:

```
{Programa que implementa o jogo Torre de Hanoi}
program TorreHanoi;
var n : integer;
    {Procedimento para indicar o movimento do disco superior de uma para outra torre}
    procedure MoveDisco(t1, t2 : string);
    begin
        writeln(t1,' → ',  t2);
    end;
    {Procedimento recursivo para a Torre de Hanoi}
    procedure Hanoi(x : integer, o, a, d : string);
    begin
        if x > 0
            then
                begin
                    Hanoi(x - 1, o, d, a);
                    MoveDisco(o, d);
                    Hanoi(x - 1, a, o, d);
                end;
    end;
{programa principal}
begin
    writeln('Digite o numero de discos ');
    readln(n);
    Hanoi(n, 'origem', 'auxiliar', 'destino');
end.
```

5.8 Exercícios propostos

1. Escreva duas funções, uma iterativa e a outra recursiva, que retornem o k-ésimo dígito (da direita para a esquerda) de um inteiro *n*. Por exemplo, K_esimoDigito(2845, 3) = 8.

2. O *fatorial ímpar* de um número *n* ímpar positivo é o produto de todos os números ímpares positivos menores do que ou iguais a *n*. Indicando o *fatorial ímpar* de *n* por n| temos, n| = 1 . 3. 5 . ... . n. Por exemplo, 7| = 1 . 3 . 5 . 7 = 105. Escreva funções iterativas e recursivas para a determinação do fatorial ímpar de um inteiro ímpar dado.

3. Como na questão anterior, o *fatorial primo* de um número primo positivo é o produto de todos os primos positivos menores do que ou iguais a ele, p# = 2 . 3 . 5 . 7 . ... .p. Por exemplo, 7# = 2 . 3 . 5 . 7 = 210. Escreva um programa que determine o fatorial primo de um primo dado.

4. Escreva uma função que retorne a soma dos algarismos de um inteiro positivo dado.

5. Escreva uma função recursiva que retorne o n-ésimo termo da seqüência de Fibbonaci, *n* dado.

6. Escreva uma função que receba um número inteiro *n* e forneça o número formado pelos algarismos de *n* escritos na ordem inversa. Por exemplo, se o número dado for 3876, a função deve fornecer 6783.

7. Escreva uma função recursiva que retorne o máximo divisor comum de dois inteiros dados.

8. Escreva uma função recursiva que retorne o mínimo múltiplo comum de dois inteiros dados.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 6 Vetores

6.1 O que são vetores

Nos exemplos 6 e 7 da seção 4.5, discutimos programas para a determinação da média aritmética de uma relação de números dados. Para tal, utilizamos uma variável simples para receber os números, sendo que cada vez que um número, a partir do segundo, era recebido, o anterior era "perdido". Ou seja, a relação de números não era armazenada. Imagine que a relação fosse uma relação de notas escolares e que, além da média, se quisesse também saber a quantidade de alunos que obtiveram nota acima da média ou uma outra medida estatística que dependesse da média (*desvio padrão*, por exemplo). Neste caso, haveria a necessidade de que a relação fosse redigitada o que, além da duplicidade do trabalho, facilitaria os erros de digitação. É importante então que exista uma "variável" capaz de armazenar vários valores simultaneamente de tal forma que se possa acessar cada um deles independentemente de se acessar os demais.

Um outro exemplo é o caso do exemplo 2 da seção 4.5. Lá queríamos a relação dos divisores de um inteiro dado e estes divisores eram apenas exibidos, não sendo armazenados como recomendado na seção 2.9. Até aquele momento, a dificuldade de se armazenar os divisores residia no fato de que não se sabe *a priori* o número de divisores de um inteiro dado e, portanto, não saberíamos quantas variáveis deveríamos declarar.

Um *vetor* é um conjunto de variáveis de um mesmo tipo as quais são acessadas através de *índices* apostos ao identificador do vetor, *índices* que devem ser de um tipo ordenado: *integer*, *char* ou *boolean* ou um *tipo definido pelo usuário*, o que veremos no capítulo 10.

Sendo um conjunto de variáveis, é comum se dizer que um vetor é um *tipo de dado estruturado*; como as componentes de um vetor são necessariamente do mesmo tipo de dado, dizemos que um vetor é *tipo de dado estruturado homogêneo*. No capítulo 8 estudaremos um *tipo de dado estruturado heterogêneo*, no sentido de um conjunto de variáveis que podem ser de tipos diferentes.

6.2 Declaração de um *vetor unidimensional*

Um *vetor unidimensional* (ou simplesmente *vetor*) é declarado através da seguinte sintaxe:

var Identificador : **array**[v_i..v_f] **of** tipo de dado;

onde o *valor inferior*, v_i, e o *valor superior*, v_f, são constantes de um tipo ordenado e limitam os possíveis valores para os índices das *componentes*, fixando também o número máximo destas componentes. Evidentemente, devemos ter $v_i \leq v_f$. De maneira mais ou menos óbvia, o *tipo de dado* da sintaxe fixará o tipo de dado das *componentes* do vetor podendo ser de qualquer tipo, inclusive um tipo estruturado.

Por exemplo, a declaração

var Vetor : **array**[1. .10] **of integer**;

definirá um conjunto de dez variáveis do tipo *integer*, enquanto que a declaração

var Cadeia : **array** [11..100] **of char**;

definirá um conjunto de noventa variáveis do tipo *char*. Por seu turno, a declaração

var Frequencia : **array**['A' .. 'Z'] **of integer**;

reservará um conjunto de vinte e seis variáveis do tipo *integer*;

Como cada variável do tipo *integer* utiliza dois bytes de memória e cada variável do tipo *char* utiliza apenas um byte, a variável *Vetor* ocupará vinte bytes de memória enquanto que a variável *Cadeia* ocupará noventa bytes.

Os compiladores Pascal, de um modo geral, possuem uma função predefinida, *SizeOf*, que retorna o número de bytes ocupado por uma variável ou por um vetor. Por exemplo, o programa

```
var x : integer;
    v : array[1..10] of real;
begin
   writeln('A variavel x ocupa ', SizeOf(x), ' bytes e v ocupa ', SizeOf(v));
end.
```

exibirá na tela a seguinte saída:

A variavel x ocupa 2 bytes e v ocupa 60 bytes

O número de bytes de *v* é explicado pelo fato de que cada variável de tipo *real* utiliza seis bytes e *v* é um conjunto de dez variáveis do tipo *real*.

Como dissemos acima, cada componente de um vetor pode ser acessada e referenciada através de índices associados ao identificador do vetor, índices que pertencem ao "intervalo" v_i .. v_f. Assim, as componentes do vetor *Cadeia* do exemplo acima serão identificadas por Cadeia[11], Cadeia[12], ..., Cadeia[100] e as componentes do vetor *Frequência* serão identificadas por Frequencia[A], Frequencia[B], ..., Frequencia[Z]. Por padrão, ocorrerá erro de compilação se algum comando do programa se referir a uma componente com um índice que não pertença ao "intervalo" v_i .. v_f.

O índice de uma componente pode ser referido através de uma expressão que resulte num valor de um tipo ordenado. Por exemplo, a seqüência de comandos e declarações

```
var i : integer;
    Quadrados : array[1 .. 100] of integer;
begin
   for i := 1 to 100 do
      Quadrados[i] := 0;
   for i := 1 to 10 do
      Quadrados[i * i] = Sqr(i);
```

gera o vetor *Quadrados* com Quadrado[1] = 1, Quadrado[4] = 4, Quadrado[9] = 9, ..., sendo iguais a zero as componentes cujas ordens não são quadrados perfeitos.

Embora os valores inicial e final v_i e v_f possam ser constantes de qualquer tipo ordenado, é mais comum que eles sejam do tipo *integer* com v_i = 1. A escolha de v_f deve ser feita através de uma estimativa do tamanho máximo do vetor, o que é possível ser feito já que o programador conhece o objetivo do programa, tendo, portanto, idéia precisa da magnitude quantitativa dos dados que o mesmo vai manipular. Estabelecida esta estimativa, pode-se definir uma constante com aquele valor e então utilizar o identificador desta constante para o limite superior. Neste caso, a utilização de uma constante para o limite superior em vez do seu próprio valor tem a vantagem de que, se a estimativa da quantidade de dados se modificar, basta se alterar a definição da tal constante.

Por exemplo, para se desenvolver um sistema para gerenciar o acervo de filmes de uma locadora de vídeos, pode-se consultar o proprietário da tal locadora em relação à quantidade de fitas que ele pretende possuir. Pode-se então, por medida de segurança, duplicar este número e definir uma constante com este valor, escrevendo o programa com declarações do tipo:

```
program locadora;
const  NumFitas = 10000;
var Titulos = array[1 .. NumFitas] of string;
```

Com estas declarações, o vetor *Titulos* pode armazenar os títulos de até 10.000 fitas. Se a locadora crescer e o acervo aumentar substancialmente, ao responsável pela manutenção do sistema caberá apenas alterar o valor da constante para, por exemplo, 15000.

6.3 Definindo um tipo vetor

Para a utilização de um vetor como um parâmetro de um subprograma, é preferível que se use um *tipo de dado definido pelo usuário*, introduzido superficialmente na seção 2.5. Isto é feito através da seguinte sintaxe

type Identificador = **array**[v_i .. v_f] **of** tipo de dado;

sendo que agora *Identificador* não será um conjunto de variáveis e sim um novo tipo de dado que pode ser utilizado para se definir variáveis de um tipo estruturado homegêneo. Por exemplo, num programa que exista a definição

```
type TVetor = array[1..10] of integer;
```

pode-se ter uma declaração do tipo

```
var Vetor : TVetor;
```

Com esta declaração *Vetor* será um conjunto de dez variáveis do tipo *integer* da mesma maneira que se tivéssemos a seguinte declaração:

```
var Vetor : array[1..10] of integer;
```

Por exemplo, o programa

```
type TVetor = array[1..100] of real;
var Vetor : TVetor;
     j : integer;
    {Procedimento para armazenar as raízes quadradas dos 100 primeiros inteiros positivos}
     procedure GeraRaizes(var v : TVetor);
     var i : integer;
     begin
          for i := 1 to 100 do
              v[i] := SqrT(i);
     end;
{Programa principal}
begin
    GeraRaizes(Vetor);
    for j := 1 to 100 do
         writeln('Raiz(', j, ') = ', Vetor[j]:0:2);
end.
```

exibe uma tabela com as raízes quadradas dos cem primeiros inteiros não negativos.

6.4 "Lendo" e "escrevendo" um vetor

Além de uma atribuição do tipo v := w, sendo *v* e *w* vetores do mesmo tipo, um vetor não pode ser referenciado globalmente. Isto significa que os comandos *readln*(v) e *write*(v) não são reconhecidos pelo sistema, se *v* for um vetor. Assim, o armazenamento de uma relação de valores num vetor e a exibição dos conteúdos das componentes de um vetor têm que ser realizadas componente a componente.

O procedimento de armazenar uma relação de dados num vetor depende do conhecimento ou não da quantidade de elementos da relação. Na hipótese de o número de elementos da relação ser conhecido, basta usar um procedimento com dois parâmetros: um para receber o vetor que vai armazenar a relação e outro para receber a quantidade de elementos da relação. Dentro do procedimento, pode-se utilizar um comando *for*.

```
{Procedimento para armazenar uma relação de dados num vetor (leitura de um vetor) quando a
quantidade de elementos da relação é conhecida}
procedure ArmazenaRelacaoN(var v : TVetor; t : integer);
var i : integer;
begin
    writeln('Digite os elementos da relacao ');
    for i := 1 to t do
         readln(v[i]);
end;
```

Se o número de elementos da relação não é conhecido *a priori*, deve-se utilizar um *flag* para encerrar a entrada dos dados, de acordo com o que foi comentado no exemplo 7 da seção 4.5. É interessante também que a função, para utilizações posteriores, determine o número de elementos da relação. Este valor pode ser

retornado através de um parâmetro com passagem por referência o qual receberá uma variável global, digamos *Tam*. Dentro da função, pode-se utilizar um comando *while* que deverá ser executado enquanto o dado de entrada for diferente do *flag* escolhido.

```
{Procedimento para armazenar uma relação de dados num vetor quando a quantidade de elementos
da relação não é conhecida}
procedure ArmazenaRelacao(var v : TVetor; var t : integer );
var i : integer;
begin
   writeln('Digite os elementos da relacao (-1 para encerrar)');
   t := 1;
   readln(v[t]);
   while (v[t] <> -1)
      begin
         t = t + 1;
         readln(v[t]);
      end;
   t := t – 1;
end;
```

Observe a necessidade do comando t := t – 1. Ele é necessário para se "excluir" o *flag*, estando o excluir entre aspas pelo fato de que o *flag* é armazenado, porém, com o comando acima, a variável que armazenará a quantidade de componentes do vetor não o considerará. Desta forma, como os acessos às componentes serão limitados ao valor desta variável, é como se o *flag* não estivesse armazenado.

Observe também a passagem de parâmetro para o parâmetro *t*: no primeiro caso, o valor da quantidade de elementos da relação era conhecido e podia ser então passado por valor; no segundo caso, o procedimento é que iria determinar esta quantidade de elementos e este valor deveria ser armazenado numa variável global.

Para se exibir uma relação de dados armazenados num vetor, basta um procedimento com dois parâmetros: um para receber o vetor onde a relação está armazenada e o outro para receber a quantidade de elementos da relação. Esta quantidade está armazenada em alguma variável, pois ela foi um dado de entrada (se ela era conhecida *a priori*) ou foi determinada na ocasião do armazenamento da relação, através de um procedimento do tipo *ArmazenaRelação* definido anteriormente.

```
{Procedimento para exibir os elementos de uma relação de dados armazenados num vetor}
procedure ExibeRelacao(var v : TVetor; t : integer);
var i : integer;
begin
   for i := 1 to t do
      write(v[i], '   ');
end;
```

Vale observar que, na execução deste procedimento, a relação está armazenada num vetor e a passagem de parâmetro para o parâmetro *v* poderia ser por valor. Porém, este tipo de passagem de parâmetro requer que fosse feita uma cópia de toda a relação que estava armazenada numa variável global para o parâmetro *v*, o que vai demandar algum tempo de processamento. Assim, quando o parâmetro é de um tipo estruturado é preferível que as passagens de parâmetros sejam feitas por referência, mesmo que isto não seja necessário.

Evidentemente o leitor deve ter entendido que um programa, para utilizar os procedimentos *ArmazenaRelacaoN*, *ArmazenaRelacao* e *ExibeRelacao* devem possuir uma definição do tipo

type TVetor = **array** [v_i .. v_f] **of** tipo de dado;

onde v_i, v_f e *tipo de dado* dependem do programa específico.

6.5 Exemplos Parte IX

Os exemplos a seguir, além de reforçar vários aspectos da utilização de vetores, são muito úteis no desenvolvimento da lógica de programação.

1. Para introduzir o estudo dos vetores, nos referimos, na seção 6.1, ao problema de se determinar o

número de alunos de uma turma que obtiveram notas maiores que a média. Com a utilização de vetores, podemos calcular a tal média e depois "percorrer" novamente o vetor, comparando cada nota com a referida média. Teríamos então o seguinte programa:

```
{Programa para determinar o numero de alunos que obtiveram notas maiores que a media}
program AvaliaNotas;
type TVetor = array [1 .. 60] of real;
var NumAlunos, NumBonsAlunos, j : integer;
     Med : real;
     RelacaoNotas : TVetor;
    {Procedimento para armazenar as notas}
    procedure ArmazenaNotas(var v : TVetor; var t : integer);
    begin
        t := 1;
        writeln('Digite as notas (-1 para encerrar)');
        readln(v[t]);
        while (v[t] <> -1) do
            begin
                t := t + 1;
                readln(v[t]);
            end;
        t := t – 1;
    end;
    {Funcao para calcular a media de uma relacao de números armazenada num vetor}
    function Media(var v : TVetor; t : integer) : real;
    var i : integer;
        Soma : real;
    begin
        Soma := 0;
        for i := 1 to t do
            Soma := Soma + v[i];
        Media := Soma/t;
    end;
{Programa principal}
begin
    ArmazenaNotas(RelacaoNotas, NumAlunos);
    Med := Media(RelacaoNotas, NumAlunos);
    NumBonsAlunos := 0;
    for j := 1 to NumAlunos do
        if (RelacaoNotas[i] > Med)
            NumBonsAlunos := NumBonsAlunos + 1;
    writeln('Media das notas:  ', Med);
    writeln('Numero de alunos com notas maiores que a media: ', NumBonsAlunos);
end;
```

2. Imaginemos agora que queiramos um função que retorne o maior valor de uma relação armazenada em um vetor. Uma possível solução é supor que o maior valor procurado é o primeiro elemento do vetor e, em seguida, percorrer o vetor comparando cada componente com o valor que até o momento é o maior, substituindo o valor deste maior elemento quando se encontra uma componente maior que ele. Por exemplo, se a relação é (8, 2, 5, 12, 1), teríamos as seguintes ações:

1. Maior := 8.
2. Como 2 < Maior, nada é feito.
3. Como 5 < Maior, nada é feito.
4. Como 12 > Maior, Maior := 12.
5. Como 1 < Maior, nada é feito.

Temos então a seguinte função:

```
{Funcao que retorna o maior elemento de uma relacao armazenada num vetor}
```

```
function MaiorElemento(var v : TVetor; t : integer) : real;
var i : integer;
    Maior : real;
begin
    Maior := v[1];
    for i := 2 to t do
        if (v[i] > Maior)
            then
                Maior := v[i];
    MaiorElemento := Maior;
end;
```

Observe que a função acima retorna o maior elemento armazenado no vetor (o que em alguns casos é suficiente), mas não retorna a posição deste maior elemento (o que em alguns casos é necessário). Para isto, podemos utilizar um procedimento com quatro parâmetros: o primeiro para receber a relação; o segundo para receber a quantidade de elementos da relação; o terceiro, por referência, para retornar o maior valor e o quarto, também por referências, para retornar a posição. Este quarto parâmetro recebe o valor um e, em seguida, o valor da posição onde foi encontrada uma componente maior do que *Maior*.

```
{Procedimento que retorna o maior valor de uma relação e a posição deste maior valor}
procedure MaiorElemento(var v : TVetor; t : integer; var Maior : real; var p : integer);
var i : integer;
begin
    Maior := v[1];
    p := 1;
    for i := 1 to t do
        if v[i] > Maior
            then
                begin
                    Maior := v[i];
                    p := i;
                end;
end;
```

Uma chamada deste procedimento vai requerer, além do vetor *Vetor* onde está armazenada a relação e do conteúdo da variável que contém o número de elementos da relação, duas outras variáveis, digamos *ElementoMaior* e *Pos*. Nestas condições a ativação do procedimento será feita através do comando

```
MaiorElemento(Vetor, Quant, ElementoMaior, Pos);
```

3. O exemplo a seguir tem o objetivo de mostrar que o índice de acesso às componentes de um vetor pode ser dado através de expressões. Nele se apresenta um procedimento que recebe um vetor e o decompõe em dois outros vetores, um contendo as componentes de ordem ímpar e o outro contendo as componentes de ordem par. Por exemplo, se o vetor dado for v = {3, 5, 6, 8, 1, 4, 2, 3, 7}, o vetor deve gerar os vetores u = {3, 6, 1, 2, 7} e w = {5, 8, 4, 3}.

```
procedure DecompoeVetor(var v, v1, v2 : TVetor; t : integer);
var i : integer;
begin
    for i := 1 to t do
        if i mod 2 = 1
            then
                v1[(i + 1) div 2] := v[i]
            else
                v2[i div 2] : = v[i];
end;
```

4. Agora apresentaremos um exemplo que mostra um vetor cujas componentes são cadeias de caracteres. Além disto, o exemplo mostra como um vetor pode ser definido como uma constante. Trata-se de uma função que retorne o nome do mês correspondente a um número dado.

```
{Funcao que retorna o nome de um mes dado}
function NomeMes(n : integer) : string;
const Mes : array [1..13] of string = ('Janeiro', 'Fevereiro', 'Marco', 'Abril', 'Maio', 'Junho', 'Julho', 'Agosto', 'Setembro', 'Outubro', 'Novembro', 'Dezembro', 'Mes invalido');
begin
   if (n > 0) and (n < 13)
      then
         NomeMes := Mes[n]
      else
         NomeMes := Mes[13];
end;
```

6.6 Vetores multidimensionais

Na seção 6.1 foi dito que um vetor é um conjunto de variáveis de mesmo tipo, chamadas componentes do vetor. A linguagem Pascal permite que as componentes de um vetor sejam também vetores, admitindo, por exemplo, uma definição tal como

```
type TMatriz = array [1 .. 50] of array [1 .. 50] of real;
     TTexto = array [1 .. 40] of array [ 1 .. 40] of array [1 .. 40] of char;
```

Naturalmente, uma variável do tipo *TMatriz* poderia armazenar uma *matriz* da matemática ou uma *tabela de dupla entrada* que, por exemplo, enumere as distâncias entre as capitais brasileiras. Já uma variável do tipo *TTexto* poderia armazenar o conteúdo de um livro considerando a página, a linha e a coluna em que cada caractere se localiza.

Para felicidade geral, as definições acima podem ser simplificadas para

```
type TMatriz = array [1 .. 50, 1 .. 50] of real;
     TTexto = array [1 .. 40, 1 .. 40, 1 .. 40] of char;
```

o que mostra que a definição de um vetor multidimensional é uma extensão natural da declaração de um vetor unidimensional.

Por exemplo, a definição e a conseqüente declaração

```
type TMatriz = array [1 .. 10, 1..8] of integer;
var Mat : TMatriz;
```

define um vetor de dez componentes, cada uma delas sendo um vetor de oito componentes, ou seja, *Mat* é um conjunto de 10 x 8 = 80 variáveis do tipo *integer*.

Para um outro exemplo, a definição/declaração

```
type TTexto = array [1 .. 72, 1 .. 30, 1 .. 30] of char;
var Livro : TTexto;
```

define uma variável *Livro* capaz de armazenar os caracteres de um livro com até setenta duas páginas, cada página possuindo trinta linhas e cada linha possuindo trinta colunas.

A referência a uma componente de um vetor multidimensional pode ser feita através de justaposição de índices dentro de colchetes ou da colocação dos índices separados por vírgulas dentro de um único par de colchetes. Assim, com os exemplos acima poderíamos ter comandos como

```
readln(Mat[1][1]);
writeln(Mat[3, 2]);
Livro[2, 5, 7} := 'X';
```

A definição de um vetor constante multidimensional segue o padrão da definição de um vetor constante unidimensional, com a ressalva de que as componentes, que agora são vetores, devem estar entre parênteses. Por exemplo, se quiséssemos um vetor bidimensional para armazenar os números de dias dos meses do ano, fazendo a distinção entre anos bissextos e não-bissextos, poderíamos declarar uma constante *DiasMeses* da seguinte forma:

const DiasMeses : **array** [1 .. 2, 1 .. 12] **of integer** = ((31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31), (31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31));

onde a primeira componente refere-se aos dias dos meses de um ano não-bissexto e a segundo, aos dias dos meses de um ano bissexto (este vetor será utilizado num exemplo a seguir).

Um vetor bidimensional é usualmente chamado de *matriz* e os números de componentes são chamados, respectivamente, *número de linhas* e *número de colunas*. Estes dois números separados por x (que é lido *por*) é a *ordem* da matriz. Assim, a variável *Mat* do exemplo acima está apta a armazenar uma matriz de ordem até 8 x 10. Estas denominações, emprestadas da matemática, são justificadas pelo fato de que, embora as componentes de um vetor bidimensional sejam armazenadas de forma consecutiva (a primeira componente do segundo vetor logo após a última componente do primeiro vetor), uma matriz, para facilitar sua compreensão, pode ser imaginada como constituída de *linhas* e de *colunas*. Por exemplo, o vetor *DiasMeses* do exemplo acima pode ser imaginado como sendo

31	28	31	30	31	30	31	31	30	31	30	31
31	29	31	30	31	30	31	31	30	31	30	31

facilitando a compreensão de referências do tipo *DiasMeses[2, 3]* que indica o elemento da segunda linha e da terceira coluna.

Se o número de linhas e o número de colunas de uma tabela são conhecidos, o seu armazenamento em uma matriz é muito simples. Basta utilizar um duplo *for* aninhado, controlados pelo número de linhas e pelo número de colunas, respectivamente.

```
{Procedimento para armazenar uma tabela de dupla entrada, com numero de linhas e número de colunas conhecidos a priori, numa matriz}
procedure ArmazenaTabelaMN(var Mat : TMatriz; m, n : integer);
var i, j : integer;
begin
   writeln('Digite, por linha, os elementos da matriz');
   for i := 1 to m do
      for j := 1 to n do
         readln(Mat[i, j]);
end;
```

Se o número de linhas e o número de colunas de uma tabela não são conhecidos, pode-se usar um duplo *while* aninhado, definindo-se um *flag* para encerramento da digitação dos elementos de cada linha e um outro *flag* para encerramento da digitação da matriz. Naturalmente, o procedimento deverá retornar o número de linhas e o número de colunas da tabela, o que justifica a passagem por referência para parâmetros *m* e *n*.

```
{Procedimento para armazenar uma tabela de dupla entrada, com numero de linhas e número de colunas não conhecidos a priori, numa matriz}
procedure ArmazenaTabela(var Mat : TMatriz; var m, n : integer);
begin
   writeln('Digite, por linha, os elementos da matriz (-1 p/ encerrar linha, -2 p/ encerrar a matriz');
   m := 1;
   n := 1;
   readln(Mat[m, n]);
   while (Mat[m, n] <> -2) do
      begin
         while (Mat[m, n] <> -1) do
            begin
               n := n + 1;
               readln(Mat[m, n]);
            end;
         m := m + 1;
         n := 0;
         readln(Mat[m, n]);
      end;
end;
```

Uma função para exibir uma tabela armazenada numa matriz também é muito simples. Basta, para que a matriz seja exibida na forma de tabela, mudar a linha cada vez que a exibição de uma linha é concluída.

```
{Procedimento para exibir uma tabela de dupla entrada armazenada numa matriz }
procedure ExibeTabela(var Mat : TMatriz; m, n : integer);
var i, j : integer;
begin
    for i := 1 to m do
        begin
            for j := 1 to n do
                write(Mat[i, j], '      ');
            writeln;
        end;
end;
```

Vale ressaltar que a passagem para o parâmetro *Mat* nos dois primeiros procedimentos é necessariamente por referência, enquanto que no terceiro só foi adotado este tipo de passagem para se evitarem cópias de conteúdos de matrizes, conforme foi observado acima.

6.7 Exemplos Parte X

1. O exemplo 2 da seção 3.8 apresentava um programa que recebia uma data e fornecia o dia da semana correspondente. Neste programa, precisamos calcular o número de dias do ano decorridos até data dada. Com a utilização da matriz *DiasMeses* comentada acima, podemos escrever facilmente uma função com este objetivo.

```
{Funcao que determina o numero de dias de um ano decorridos ate uma data dada}
function DiasAno(d, m, a : integer) : integer;
const DiasMeses : array [1 .. 2, 1 ..12] of integer = ((31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31),
(31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31));
var i : integer;
begin
    if ((a mod 4 = 0) and (a mod 100 <> 0)) or (a mod 400 = 0)
        then
            for i := 2 to m do
                d := d + DiasMeses[2, i - 1]
        else
            for i := 2 to m do
                d := d + DiasMeses[1, i - 1];
    DiasAno := d;
end;
```

2. Como no exemplo em que o próprio sistema gerou as raízes quadradas dos cem primeiros números inteiros positivos, o sistema pode gerar uma matriz. Para exemplificar isto, apresentaremos um procedimento que gera a *matriz identidade de ordem n*. Para um inteiro positivo dado, a *matriz identidade de ordem n* é a matriz $I_n = (i_{rs})$, de ordem nxn, dada por $i_{rs} = 1$, se r = s, e $i_{rs} = 0$, se $r \neq s$. Esta matriz é muito importante no estudo das matrizes, sendo utilizada, por exemplo, para a determinação da *matriz inversa* de uma matriz inversível. Por exemplo, se n = 3, temos

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
{Procedimento que gera a matriz unidade de ordem n}
procedure GeraMatrizUnidade(var Mat : TMatriz; n : integer);
var i, j : integer;
begin
    for i := 0 to n do
```

```
            for j = 0 to m do
                if i = j
                    then
                        Mat[i, j] = 1
                    else
                        Mat[i, j] = 0;
    end;
```

3. Quando o número de linhas de uma matriz é igual ao número de colunas a matriz é dita *matriz quadrada*. Neste caso, os elementos de índices iguais constituem a *diagonal principal*. A soma dos elementos da diagonal principal de uma matriz quadrada é o *traço* da matriz. Como mais um exemplo de programas que manipulem matrizes, a função abaixo determina o *traço* de uma matriz quadrada dada. Observe que, para percorrer a diagonal principal, não há necessidade de um duplo *for*.

```
{Funcao que calcula o traco de uma matriz}
function Traco(var Mat : matriz; m, n : integer) : real;
var i : integer;
    Tr : real;
begin
  if  m = n
      then
        begin
          Tr := 0;
          for i := 1  to m  do
            Tr := Tr + Mat[i, i];
          Traco := Tr;
        end;
      else
        writeln('A  matriz não é quadrada');
end;
```

4. Uma tabela que enumere as distâncias entre várias cidades é uma matriz *simétrica*: os termos simétricos em relação à *diagonal principal* são iguais, ou seja, Mat[i][j] = Mat[j][i]. Obviamente a digitação de uma matriz com esta propriedade pode ser simplificada, devendo-se digitar apenas os termos que estão acima da diagonal principal.

```
{Procedimento para armazenar uma matriz simetrica}
procedure ArmazenaMatrizSimetrica(var Mat : TMatriz; m : integer);
var i, j : integer;
begin
    writeln('Digite, por linha, os elementos da matriz, a partir da diagonal');
    for i := 0 to m do
        for j := i to m do
            begin
                readln(Mat[i][j]);
                Mat[j][i] := Mat[i][j];
            end;
end;
```

Observe a inicialização de *j* no comando *for*. A razão disto é que só serão digitados os termos acima da diagonal principal, termos em que $j \geq i$. Observe também que estamos utilizando a outra forma de se referenciar as componentes de uma matriz.

5. Nos exemplos anteriores, sempre "percorremos a matriz pelos elementos de suas linhas", que é o usual. O próximo exemplo mostra um caso em que é necessário percorrer as colunas. Trata-se de um procedimento para uma questão muito comum: a totalização das colunas de uma tabela.

```
{Procedimento para totalizar as colunas de uma tabela armazenada numa matriz}
procedure TotalizaColunas(var Mat : TMatriz; m, n : integer);
var i, j : integer;
begin
```

```
        for j := 0 to n do
            begin
                Mat[m + 1][j] := 0;
                for i := 0 to m do
                    Mat[m + 1][j] := Mat[m + 1][j] + Mat[i][j];
            end;
    end;
```

Observe que a totalização das colunas foi armazenada na linha de ordem $m + 1$, onde m é o número de linhas da tabela. Observe também a necessidade de, para cada j, inicializar-se Mat[m + 1][j] com zero.

6. O exemplo a seguir, além de sua importância prática, tem o objetivo de mostrar que se pode utilizar definições específicas para vetores multidimensionais de acordo com o objetivo do programa. Trata-se de um programa que determine as médias das linhas de uma matriz. Por exemplo, se a matriz dada for $\begin{pmatrix} 3 & 7 & 4 & 6 \\ 5 & 4 & 5 & 4 \\ 2 & 6 & 5 & 1 \end{pmatrix}$ a função deve fornecer a matriz $\begin{pmatrix} 3 & 7 & 4 & 6 & 5{,}0 \\ 5 & 4 & 5 & 4 & 4{,}5 \\ 2 & 6 & 5 & 1 & 3{,}5 \end{pmatrix}$. Este programa que poderia ser utilizado para, dada uma tabela com os preços de vários produtos em vários supermercados, determinar o preço médio de cada um dos produtos. Como precisamos calcular a média de cada linha e sabemos calcular a média das componentes de um vetor, vamos definir a matriz como sendo, especificamente, um vetor cujas componentes são vetores:

```
{Programa para determinar as medias de cada uma das linhas de uma matriz}
program MediasdasLinhas;
type TVetor = array[1..20] of real;
     TTabela = array[1..50] of TVetor;
var Tabela : TTabela;
    NumLinhas, NumColunas : integer;
        {Procedimento para armazenar uma tabela}
        procedure ArmazenaTabela(var Tab : TTabela; m, n : integer);
        var i, j : integer;
        begin
            writeln('Digite (por linha) os elementos da tabela');
            for i := 1 to m do
                for j := 1 to n do
                    readln(Tab[i, j]);
        end;
        {Procedimento para exibir uma tabela}
        procedure ExibeTabela(var Tab : TTabela; m, n : integer);
        var i, j : integer;
        begin
            for i := 1 to m do
                begin
                    for j := 1 to n do
                        write(Tab[i, j]:6:2);
                    writeln;
                end;
        end;
        {Funcao para calcular a media das componentes de um vetor}
        function Media(var v : TVetor; t : integer) : real;
        var Soma : real;
            i : integer;
        begin
            Soma := 0;
            for i := 1 to t do
                Soma := Soma + v[i];
            Media := Soma/t;
```

```
        end;
        {Procedimento para determinar a media de cada uma das linhas de uma matriz}
        procedure MediaLinha(var Tab : TTabela; m, n : integer);
        var i : integer;
        begin
          for i := 1 to m do
            Tab[i, n + 1] := Media(Tab[i], n);
        end;
    {Programa principal}
    begin
      writeln('Digite o numero de linhas e o numero de colunas da tabela');
      readln(NumLinhas, NumColunas);
      ArmazenaTabela(Tabela, NumLinhas, NumColunas);
      MediaLinha(Tabela, NumLinhas, NumColunas);
      ExibeTabela(Tabela, NumLinhas, NumColunas + 1);
    end.
```

Observe que cada linha de *Tabela* é um vetor do tipo *TVetor* e, portanto, pudemos passar cada linha para a função *Media*.

6.8 Um programa para medidas estatísticas

Estamos em condições de apresentar um "programa completo", com um objetivo específico. Trata-se de um programa que determina medidas estatísticas, utilizadas em problemas de economia, de administração, de educação e de vários outros ramos do conhecimento humano. Estas medidas objetivam avaliar uma relação de valores numéricos relativos a algum fenômeno, procurando medir valores para os quais os valores da relação tendem e como estes valores estão dispersos. As medidas que o programa determina são a *média aritmética* e a *moda*, que são *medidas de tendência central*, e a *amplitude* e o *desvio padrão*, que são *medidas de dispersão*. A *média aritmética* (já discutida nos exemplos 1 da seção 6.5, 6 e 7, da seção 4.5 e 2, da seção 1.7) é definida como o quociente entre a soma dos valores da relação e a quantidade de seus elementos; a *moda* é o valor de maior freqüência da relação; a *amplitude* é a diferença entre o maior e o menor valores da relação; o *desvio padrão* é a raiz quadrada da média aritmética dos quadrados dos *desvios em relação à média*, que são as diferenças entra cada valor e a média aritmética da relação.

Para discutir as funções que farão parte do programa, imaginemos que a relação esteja armazenada num vetor *Relacao*. Para o cálculo da moda, vamos construir um vetor *Frequencia*, com número de componentes igual ao número de componentes do vetor *Relacao*, para armazenar as freqüências de cada um dos valores da relação. Para isto, inicializamos cada componente *Frequencia*[i] com zero e percorremos o vetor *Relacao*, a partir da posição i, procurando valores iguais à *Relacao*[i]; quando um valor igual é encontrado, a componente *Frequencia*[i] é incrementada. Em seguida, determinamos a posição *p* do maior elemento do vetor *Frequencia*; a *moda* da relação é, então, *Relacao*[p].

O cálculo da *amplitude* é bem mais simples: bastam funções que calculem a maior e a menor componente do vetor *Relacao* e a amplitude é a diferença entre os valores retornados por estas funções. Para que a função que determina o maior elemento possa ser utilizada também no cálculo da moda, ela deve retornar também a posição deste maior elemento, como foi feito no exemplo 2 da seção 6.5.

Para a determinação do *desvio padrão*, construímos um vetor contendo os quadrados dos *desvios em relação à média*, calculamos a média aritmética das componentes deste vetor e, finalmente, a raiz quadrada desta média.

Observe que este programa exemplifica o que foi citado na seção 5.1 a respeito da utilização de subprogramas: um subprograma é utilizado por outros subprogramas dentro de um mesmo programa.

Cabe ressaltar também que existe uma outra medida de tendência central importante chamada *mediana*. Ela não foi incluída no exemplo pelo fato de que o seu cálculo fica facilitada se a tabela estiver ordenada e só discutiremos ordenação das componentes de um vetor num próximo capítulo.

```
{Programa para determinar medidas estatisticas: media, amplitude, moda, desvio padrao}
program MedidasEstatisticas;
uses Crt;
type TVetor = array [1 .. 500] of real;
var Relacao : TVetor;
    Tam : integer;
    Opcao, OutraMedida, OutraRelacao : char;
        {Procedimento para geração de um menu de opcoes}
        procedure Menu;
        begin
            writeln('     1-Media');
            writeln('     2-Amplitude');
            writeln('     3-Moda');
            writeln('     4-Desvio Padrao');
            writeln('                  Digite sua opcao: ');
        end;
        {Procedimento para armazenar uma relacao de números num vetor}
        procedure ArmazenaRelacao(var v : TVetor; var t : integer );
        var i : integer;
        begin
            writeln('Digite os elementos da relacao (-1 para encerrar)');
            t := 1;
            readln(v[t]);
            while (v[t] <> -1) do
                begin
                    t := t + 1;
                    readln(v[t]);
                end;
            t := t - 1;
        end;
        {Funcao para calcular a media de uma relacao de números armazenada num vetor}
        function Media(var v : TVetor; t : integer) : real;
        var i : integer;
            Soma : real;
        begin
            Soma := 0;
            for i := 1 to t do
                Soma := Soma + v[i];
            Media := Soma/t;
        end;
        {Funcao que retorna o maior elemento de uma relacao armazenada num vetor e a sua posicao
no vetor}
        function MaiorElemento(var v : TVetor; t: integer; var Pos: integer) : real;
        var i : integer;
            Maior : real;
        begin
            Maior := v[1];
            Pos := 1;
            for i := 1 to t do
                if v[i] > Maior
                    then
                        begin
                            Maior := v[i];
                            Pos := i
                        end;
            MaiorElemento := Maior;
        end;
```

```
        {Funcao que retorna o menor elemento de uma relacao armazenada num vetor}
        function MenorElemento(var v : TVetor; t : integer) : real;
        var i : integer;
            Menor : real;
        begin
            Menor := v[1];
            for i := 1 to t do
                if v[i] < Menor
                    then
                        Menor := v[i];
            MenorElemento := Menor;
        end;
        {Funcao que determina a amplitude de uma relacao armazenada num vetor}
        function Amplitude(var v : TVetor; t : integer) : real;
        var p : integer;
            Maior, Menor : real;
        begin
            Maior := MaiorElemento(v, t, p);
            Menor := MenorElemento(v, t);
            Amplitude := Maior - Menor;
        end;
        {Funcao que retorna a moda de uma relacao armazenada num vetor}
        function Moda(var v : TVetor; t : integer) : real;
        var Frequencia : TVetor;
            i, j, p : integer;
        begin
            for i := 1 to t do
                begin
                    Frequencia[i] := 0;
                    for j := i to t do
                        if v[j] = v[i]
                            then
                                Frequencia[i] := Frequencia[i] + 1;
                end;
            MaiorElemento(Frequencia, t, p);
            Moda := v[p];
        end;
        {Funcao para o calculo do desvio padrao}
        function DesvioPadrao(var v : TVetor; t : integer): real;
        var d : TVetor;
            i : integer;
            m : real;
        begin
            m := Media(v, t);
            {Determinacao dos quadrados dos desvios}
            for i := 1 to t do
                d[i] := Sqr(v[i] - M);
            DesvioPadrao := SqrT(Media(d, t));
        end;
{Programa principal}
begin
    repeat
        ArmazenaRelacao(Relacao, Tam);
        repeat
            Menu;
            readln(Opcao);
```

```
            Case Opcao of
               '1' : writeln('Media : ', Media(Relacao, Tam));
               '2' : writeln('Amplitude : ', Amplitude(Relacao, Tam));
               ´3' : writeln('Moda : ', Moda(Relacao, Tam));
               '4' : writeln('Desvio Padrao : ', DesvioPadrao(Relacao, Tam));
            end;
            Delay(1000);
            write('Outra medida (S/N)? ');
            readln(OutraMedida);
         until UpCase(OutraMedida) = 'N';
         write('Analisa outra relacao (S/N)? ');
         readln(OutraRelacao);
      until UpCase(OutraRelacao) = 'N';
   end.
```

6.9 Exercícios propostos

1. Escreva um procedimento que exiba as componentes de um vetor na ordem inversa daquela em que foram armazenadas.

2. Um vetor é *palíndromo* se ele não se altera quando as posições da componentes são invertidas. Por exemplo, o vetor v = {1, 3, 5, 2, 2, 5, 3, 1} é palíndromo. Escreva uma função que verifique se um vetor é palíndromo.

3. Escreva um procedimento que, recebendo dois vetores com a mesma quantidade de elementos, gere um outro vetor intercalando as componentes dos vetores dados. Por exemplo, se v_1 = {4, 8, 1, 9} e v_2 = {2, 5, 7, 3} o procedimento deve gerar o vetor v = {4, 2, 8, 5, 1, 7, 9, 3}.

4. Escreva um procedimento que decomponha um vetor de inteiros em dois outros vetores, um contendo as componentes de valor ímpar e o outro contendo as componentes de valor par. Por exemplo, se o vetor dado for v = {3, 5, 6, 8, 1, 4, 2, 3, 7} o procedimento deve gerar os vetores u = {3, 5, 1, 3, 7} e w = {6, 8, 4, 2}.

5. Um *vetor* do R^n é uma n-upla de números reais v = {x_1, x_2, ..., x_n}, sendo cada x_i chamado de *componente*. A *norma* de um vetor v = {x_1, x_2, ..., x_n} é definida por $\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$. Escreva uma função que receba um vetor do R^n, n dado, e forneça sua norma.

6. O *produto escalar* de dois vetores do R^n é a soma dos produtos das componentes correspondentes, isto é, se u = {x_1, x_2, ..., x_n} e v = {y_1, y_2, ..., y_n}, o *produto escalar* é $x_1.y_1 + x_2.y_2 \ldots + x_n.y_n$. Escreva uma função que receba dois vetores do R^n, n dado, e forneça o produto escalar deles.

7. Escreva um procedimento que forneça as componentes distintas de um vetor dado. Por exemplo, se o vetor dado for v = {3, 2, 1, 3, 4, 1, 5, 5, 2}, o procedimento deve fornecer v = {3, 2, 1, 4, 5}.

8. O exemplo 2 da seção 2.9 pedia uma função para extrair o algarismo da casa das unidades de um inteiro dado. Aparentemente esta questão não tem interesse prático. Vejamos um problema cuja solução dependa desta questão. Algumas empresas que realizam sorteios de prêmios entre seus clientes o fazem através dos sorteios da loteria federal, sendo ganhador o número formado pelos algarismos das casas das unidades dos números sorteados no cinco prêmios da referida loteria. Por exemplo, se o sorteio da loteria federal deu como resultado os números 2345**1**, 0023**4**, 1123**6**, 0123**5** e 2345**2**, o prêmio da tal empresa seria dado ao cliente que possuísse o bilhete de número **14652**. Escreva uma função que receba os números sorteados pela loteria federal e forneça o número que ganhará o prêmio de acordo com as regras acima.

9. Escreva um procedimento que insira um valor dado num vetor numa posição dada. Por exemplo, se o vetor for v = {3, 8, 5, 9, 12, 3}, o valor dado for 10 e a posição dada for 4, o procedimento deve gerar v = {3, 8, 5, **10**, 9, 12, 3}.

10. Escreva um procedimento que insira um valor dado num vetor ordenado de modo que o vetor continue ordenado. Por exemplo, se o vetor dado for v = {2, 5, 7, 10, 12, 13} e o valor dado for 6, o procedimento deve fornecer o vetor v = {2, 5, **6**, 7, 10, 12, 13}.

11. Escreva um procedimento que delete uma componente de ordem dada de um vetor dado. Por exemplo, se o vetor dado for v = {2, 5, 7, 10, 12, 13} e a componente a ser deletada for a de ordem 4, o procedimento deve fornecer o vetor v = {2, 5, 7, 12, 13}.

12. Escreva um procedimento que, dadas duas relações de números, cada uma delas com números distintos, forneça os números que aparecem nas duas listas. Por exemplo, se as relações forem u = {9, 32, 45, 21, 56, 67, 42, 55} e w = {24, 42, 32, 12, 45, 11, 67, 66, 78}, o procedimento deve fornecer o vetor v = {32, 45, 67, 42}.

13. Escreva um procedimento que, dado um vetor ordenado, forneça a maior diferença entre duas componentes consecutivas, fornecendo também as ordens das componentes que geraram esta maior diferença. Por exemplo, se o vetor dado for v = {3, 5, 9, 16, 17, 20, 26, 31}, o procedimento deve fornecer como maior diferença o valor 7 (16 - 9), e as ordens 4 e 3.

14. Uma avaliação escolar consiste de 50 questões objetivas, cada uma delas com 5 opções, v = {1, 2, 3, 4 e 5}, sendo apenas uma delas verdadeira. Escreva um procedimento que receba a seqüência de respostas corretas, o *gabarito*, e corrija um cartão-resposta dado.

15. Escreva um programa que forneça o valor numérico de um polinômio P(x) dado, para um valor de x dado. Por exemplo, se o polinômio dado for $P(x) = x^3 + 2x - 1$ e o valor de x dado for 2, o programa deve fornecer $P(2) = 2^3 + 2x2 - 1 = 11$.

16. A matemática prova que a conversão de um número do sistema decimal para o sistema binário pode ser feita através de divisões sucessivas do número e dos quocientes sucessivamente obtidos por 2, sendo então o número binário dado pela seqüência iniciada por 1 e seguida pelos restos obtidos nas divisões sucessivas, na ordem inversa em que são obtidos. Por exemplo, para se converter 22 do sistema decimal para o sistema binário, temos: 22 **mod** 2 = 0; 11 **mod** 2 = 1; 5 **mod** 2 = 1; 2 **mod** 2 = 0 e, portanto, $22 = (10110)_2$. Escreva uma função que converta um número positivo dado no sistema decimal de numeração para o sistema binário, usando o algoritmo acima.

17. O exercício 10 da seção 4.6 solicitava um programa que determinasse a *decomposição em fatores primos*, fornecendo os fatores primitivos e suas respectivas *multiplicidades*. Na ocasião, os fatores primos e suas multiplicidades eram apenas exibidos não sendo armazenados. Modifique a função referida para que os fatores primos e as suas multiplicidades sejam armazenados antes de serem exibidos.

18. A Universidade Federal de Alagoas (UFAL) adota o sistema de verificação de aprendizagem listado no exemplo 8 da seção 3.5, com o adendo de que terá direito a uma *reavaliação* um aluno que obtiver uma nota inferior a 7,0 em algum bimestre. Neste caso, a nota obtida na reavaliação substitui a menor das notas bimestrais obtidas. Escreva um procedimento que, recebendo as notas das avaliações bimestrais e, se for o caso, a nota da reavaliação e, se for o caso, a nota da prova final, forneça a média final de um aluno da UFAL e a sua condição em relação à aprovação.

19. Escreva um procedimento que forneça a *transposta* de uma matriz dada.

20. Um dos métodos para se estudarem as soluções de um *sistema linear de n equações a n incógnitas* aplica *operações elementares sobre as linhas da matriz dos coeficientes*, sendo a permuta de duas linhas uma destas operações elementares. Escreva um procedimento que permute as posições de duas linhas de uma matriz dadas.

21. Uma matriz quadrada é dita *triangular* se os elementos situados acima de sua diagonal principal são todos nulos. Escreva uma função que receba uma matriz quadrada e verifique se ela é *triangular*.

22. O exemplo 4 da seção 6.6 apresentou um procedimento para armazenar uma matriz simétrica. Este exercício quer algo contrário: escreva uma função que verifique se uma matriz dada é simétrica.

23. Escreva um procedimento que determine o produto de duas matrizes.

24. Escreva um programa que determine o menor valor de cada uma das linhas de uma matriz dada, fornecendo o índice da coluna que contém este menor valor. Por exemplo, se a matriz dada for

$$\begin{pmatrix} 3 & 7 & 4 & 6 \\ 5 & 2 & 5 & 4 \\ 2 & 6 & 5 & 1 \end{pmatrix}$$

, a função deve fornecer uma tabela do tipo

Linha	Menor valor	Coluna
1	3	1
2	2	2
3	1	4

Um programa como este poderia receber os preços de diversos produtos praticados por vários supermercados e forneceria, para cada produto, o menor preço e o supermercado que pratica este melhor preço.

25. No exemplo 4 da seção anterior, vimos como armazenar uma matriz simétrica. Na prática, uma matriz deste tipo ocorre, por exemplo, numa tabela de distâncias entre cidades, como a seguinte tabela que dá as distâncias aéreas, em Km, entre as capitais dos estados nordestinos (Aracaju, Fortaleza, João Pessoa, Maceió, Natal, Recife, Salvador, São Luís, Teresina).

	A	F	JP	M	N	R	S	SL	T
A	0	812	438	210	550	398	267	1218	1272
F	812	0	562	730	444	640	1018	640	432
JP	418	562	0	284	144	110	758	1208	987
M	210	730	294	0	423	191	464	1220	1126
N	550	414	144	423	0	252	852	1064	843
R	398	640	118	191	252	0	654	1197	935
S	267	1018	758	464	852	654	0	1319	1000
SL	1218	640	1208	1220	1064	1197	1319	0	320
T	1272	432	987	1126	843	935	1000	320	0

Imagine que uma companhia de transporte aéreo estabeleça que uma viagem entre duas cidades que distem mais de 400 Km deve ter uma escala. Escreva um programa que armazene uma tabela das distâncias aéreas entre n cidades e dadas duas cidades determine, se for o caso, a cidade em que deve se realizar uma escala para que o percurso seja o menor possível. Por exemplo, nas condições estabelecidas, a viagem entre Maceió e São Luís deve ter uma escala em Fortaleza (o percurso Maceió/Fortaleza/São Luís é de 1370 Km; o percurso, por exemplo, Maceió/Recife/São Luís é de 1388 Km).

26. (Problema não-trivial) Utilizando uma função recursiva, construa um programa que escreva as combinações dos números 1, 2, ..., n, com taxa *k*, *n* e *k* dados. Por exemplo, se n = 5 e k = 3, o programa deve exibir

1, 2, 3

1, 2, 4

1, 2, 5

1, 3, 4

1, 3, 5

1, 4, 5

2, 3, 4

2, 3, 5

2, 4, 5

3, 4, 5.

Capítulo 7 Cadeia de Caracteres (*Strings*)

7.1 O que são cadeias de caracteres

Como foi dito na seção 5.6, Pascal possui um "tipo de dado", denominado *string*, capaz de armazenar uma cadeia de caracteres. Na verdade, uma *string* pode ser vista como um vetor cujas componentes são variáveis do tipo *char*, com a diferença que uma *string* pode ser acessada globalmente. Além disto, o sistema acrescenta uma componente de índice zero que armazena informações sobre o número de caracteres da *string*. Este número de caracteres é, naturalmente, chamado de *comprimento da string.*

A declaração de um *string* deve ser feita através da seguinte sintaxe:

var Identificador : **string**[n];

onde *n* define o comprimento máximo de uma cadeia de caracteres que pode ser armazenada na variável. O sistema limita este número a 255 e quando se pretende que a *string* tenha este tamanho máximo o parâmetro *n* e os colchetes podem ser omitidos. Estes elementos também são omitidos numa definição de um tipo de dado de um parâmetro de um subprograma ou do tipo de uma função.

Como dissemos acima, além das componentes reservadas para o armazenamento dos caracteres, o sistema reserva uma outra componente, de índice 0 (zero), para guardar informações a respeito do comprimento corrente da *string*. Estas informações são armazenadas através do caractere cujo Código ASCII Decimal é o número atual de caracteres armazenados.

Por exemplo, uma declaração do tipo *var* St : *string*[8]; faz com que o sistema reserve 9 (nove) variáveis do tipo *char*, deixando a variável *St* apta a armazenar cadeias com até 8 (oito) caracteres. Armazenando-se, por exemplo, a cadeia *Livro* em *St*, teremos a seguinte disposição:

Chr(5)	L	i	v	r	o			

Neste exemplo, St[1] = L, St[4] = r, etc. e St[0] = *Chr*[5] e, portanto, o comprimento da cadeia armazenada pode ser obtido através de *Ord*(*St*[0]). Na verdade, o sistema oferece uma função predefinida *Length*(x) que fornece o comprimento da cadeia armazenada na variável *x*. Naturalmente, *Length*(x) = *Ord*(x[0]).

Tentativas de se armazenarem cadeias com um número maior de caracteres do que o número máximo definido para a variável não provocam erros de compilação nem de execução. Apenas os caracteres excedentes serão perdidos. Se tentássemos armazenar a palavra *Matematica* na variável *St* do exemplo anterior, os caracteres *c* e *a* seriam perdidos e a disposição seria a seguinte:

Chr(8)	M	a	t	e	m	a	t	i

Além da possibilidade de acesso a cada componente da *string*, através de índices, uma variável deste tipo pode ser acessada globalmente em comandos de entrada, de saída e de atribuição. Além disso, pode-se realizar comparações entre os conteúdos de variáveis ou entre variáveis e constantes deste tipo.

Por exemplo, o programa abaixo é um programa correto e sua saída depende dos valores fornecidos nos comandos de entrada.

```
program ExemploStrings;
var St1, St2, St3 : string[20];
begin
   readln(St1);
   readln(St2);
   St3 := 'LUA'
   if (St1 < St2) or ('SOL' < St3)
      then
         writeln(St1,'     ',St2)
      else
         writeln(St3);
end;
```

Como mostra o comando St3 := 'LUA', referências a constantes do tipo *string*, são feitas com elas escritas entre apóstrofos. Isto é necessário para que o sistema saiba que se trata de uma constante e não de um identificador de variável.

7.2 Exemplos Parte XI

1. A função abaixo utiliza a função predefinida *UpCase* para retornar uma *string* com todas as suas letras escritas na forma de letras maiúsculas.

```
function CaixaAlta( s : string ) : string;
var i : integer;
begin
    for i := 1 to Length(s) do
        s[i] := UpCase(s[i]);
    CaixaAlta := s;
end;
```

2. O programa abaixo determina, de maneira evidente, o número de vogais de uma frase.

```
program ContaVogais;
var Frase : string;
    i, ContVog : integer;
begin
    writeln('Digite a frase');
    readln(Frase);
    ContVog := 0;
    for i := 1 to Length(Frase) do
        case UpCase(Frase[i]) of
            'A','E','I','O','U' : ContVog := ContVog + 1;
    writeln('A frase: ', Frase,' possui ', ContVog, ' vogais ');
end.
```

Observe que o uso da função *UpCase* serve apenas para diminuir número de comparações, pois cada caractere de *Frase* é comparado apenas com as vogais maiúsculas. Talvez as sucessivas chamadas a esta função prejudique a eficiência do programa, porém, dentro de parâmetros razoáveis, estamos dando preferência à clareza em relação à eficiência.

Observe também que a utilização do comando *case* em vez do comando *if* simplifica muito o programa, pois se o *if* fosse utilizado haveria necessidade de se escrever expressão lógica envolvendo 5 (cinco) relações (na verdade, isto poderia ser evitado com o uso do tipo *set* que mostraremos no capítulo 10).

3. O programa abaixo apresenta um exemplo de um vetor indexado por uma variável do tipo *char*. Seu objetivo é determinar as freqüências de cada letra do alfabeto em um texto dado, questão que foi muito utilizada em decodificações de mensagens codificadas por métodos antigos.

A idéia é utilizar um vetor de inteiros, indexado pelas letras do alfabeto, sendo cada componente deste vetor o contador para letra que indexa a tal componente. Para trabalhar apenas com letras maiúsculas, chamamos a função *CaixaAlta* apresentada acima, que supomos fazer parte do programa.

```
program ContaLetras;
type TVetor = array['A'..'Z'] of integer;
var Texto, Aux : string;
    ContaLetra : TVetor;
    l : char;
    i : integer;
begin
    for l := 'A' to 'Z' do
        ContaLetra[l] := 0;
    writeln('Digite o texto');
    readln(Texto);
```

```
        Aux := CaixaAlta(Texto);
        for i := 1 to Length(Texto) do
            ContaLetra[Aux[i]] := ContaLetra[Aux[i]] + 1;
        for l := 'A' to 'Z' do
            writeln( l,'   ', ContaLetra[L]);
    end.
```

Observe que a primeira estrutura de repetição inicializa as componentes do vetor *ContaLetra* com 0 (zero), enquanto que a segunda percorre todo o *Texto* contando cada uma das suas letras através da componente de *ContaLetra* indexada pela letra respectiva.

4. O exemplo a seguir apresenta um procedimento que deleta de uma *string* uma quantidade dada de caracteres, a partir de uma posição dada.

```
procedure Deleta(var s : string; i, n : integer);
var j, c : integer;
begin
    c := Length(s);
    for j := i to c  do
        s[j] := s[j + n]
    if i + n <= c
        then
            s[0] := Chr(c - n)
        else
            if  i < c
                then  s[0] := Chr(i);
end;
```

Observe que optamos por armazenar o comprimento da *string* numa variável. Isso foi feito para evitar várias chamadas à função *Length*, já que este comprimento seria utilizado em várias partes do procedimento. Observe também a atribuição feita em s[0]. Isto foi feito para alterar o comprimento da *string*.

Outra observação importante é que, como veremos abaixo, existe uma função predefinida que realiza deleções em *strings*. O exemplo acima seria o que chamamos de uma *implementação* de uma função predefinida.

7.3 Funções e procedimento predefinidos para manipulação de cadeias de caracteres

Além da função *Length* que determina o comprimento de uma *string*, os compiladores de Pascal oferecem alguns procedimentos e algumas funções que facilitam o projeto de programas que manipulam cadeias de caracteres. Estes procedimentos e estas funções pertencem à unidade *System* e estão sempre disponíveis, pois, como já foi observado anteriormente, esta unidade é carregada junto com o sistema.

a) Concatenando *strings*.

A concatenação de cadeias de caracteres pode ser feita de duas formas. A primeira delas através da função

Concat(s1, s2, ... , sn : **string**) : **string**;

e a segunda através do operador **+**

Por exemplo, se *St*, *St1* e *St2* são variáveis do tipo *string*, a seqüência de comandos

```
St1 := 'Computa';
St2 := 'dor';
St := Concat(St1, St2);
```

armazena em *St* a *string Computador*; efeito que também seria obtido através da seqüência

```
St1 := 'Computa';
St2 := 'dor';
```

St := St1 + St2;

b) Verificando se uma *string* é *substring* de uma outra *string*.

De um modo geral, os processadores de texto oferecem recursos que permitem localizar uma palavra em um texto, ou seja, permitem verificar se uma palavra está contida em um texto. Pascal possui a função

Pos(St1, St2 : **string**) : **byte**;

que verifica se a *string St1* está contida na *string St2*, retornando a posição a partir da qual isto ocorre ou zero se *St1* não é *substring* de *St2*. Por exemplo, se *p* é uma variável do tipo *integer*, o comando

p := ('Federal', 'Universidade Federal de Alagoas');

armazena em *p* o valor 14, enquanto que o comando

p := ('Fedetal', 'Universidade Federal de Alagoas');

armazena em *p* o valor zero.

c) Extraindo uma *substring* de uma *string*

Em algumas situações, há a necessidade de que uma parte de uma cadeia de caracteres seja armazenada numa outra variável. Isto pode ser feito através da execução da função

Copy(s : **string**, i, n : **integer**) : **string**;

que retorna, da *string* s, a *substring* com *n* caracteres, a partir do índice *i*. Por exemplo, se *Data* e *Mes* são variáveis do tipo *string*, a seqüência de comandos

Data := '07/09/1822';
Mes := **Copy**(Data, 4, 2);

armazena em *Mes* a cadeia de caracteres *09*.

Se o valor do parâmetro *i* for maior que o comprimento da *string s*, a função *Copy* retorna a *string vazia* (*string* de comprimento zero) e se *i + n* é maior que o citado comprimento, a função retornará a *substring* composta dos últimos caracteres de *s*, a partir de *i*. Por exemplo, se *St* é uma *string* o comando

St := **Copy**('Computacao', 7, 10);

armazenará em *St* a *string acao*.

d) Excluindo caracteres de uma *string*

A exclusão de um dado número de caracteres de uma *string* pode ser feita através da execução do procedimento

Delete(**var** s : **string**; i, n : **integer**);

que exclui, da *string s*, *n* caracteres a partir do índice *i*. Naturalmente, se *i* é maior que o comprimento da *string*, nenhuma exclusão é realizada e se *i + n* é maior que o referido comprimento, apenas os últimos caracteres, a partir de *i* são deletados.

Exemplificando: se o conteúdo da variável St é *CAMINHAO*, a chamada de *Delete*(St, 3, 2) altera o conteúdo de *St* para *CANHAO*; a chamada de *Delete*(St, 9, 1) não altera o conteúdo de *St* enquanto que a ativação de *Delete*(St, 8, 5) altera o conteúdo de *St* para *CAMINHA*. Observe que o parâmetro *s* recebe os argumentos por referência, o que significa que ele receberá uma variável e é o conteúdo desta variável que, se for o caso, será alterado.

e) Inserindo uma *string* em outra

Pode-se inserir uma *string* numa cadeia de caracteres através do procedimento

Insert(St1 : **string**; **var** St2 : **string**; i : **integer**);

que insere a *string St1* na *string St2*, a partir do índice *i*. Como a passagem do argumento para *St2* é por referência, este argumento será alterado com a inserção realizada. Assim, se o conteúdo da variável *St* é *ANTA*, a chamada de *Insert*('UME', St, 2) altera o conteúdo de St para AUMENTA.

f) Convertendo, se possível, uma *string* em um número

Uma cadeia de caracteres constituída de dígitos, precedidos ou não por um dos caracteres + ou -, e de, no máximo, um ponto, pode ser convertida numa constante do tipo *integer* (se a cadeia não possuir um ponto) ou numa constante do tipo *real* (se um dos caracteres da cadeia for um ponto). Isto é feito com a execução do procedimento

```
Val(s : string; var v : integer/real; var r : integer);
```

sendo que o argumento passado para o parâmetro *r* recebe 0 (zero) se a conversão foi bem sucedida ou o índice do primeiro caractere não permitiu a conversão. É claro que a conversão não será bem sucedida se o argumento passado para *s* contiver um caractere que não seja um dígito, um ponto decimal ou **+/-** no seu início. Este procedimento pode ser usado, por exemplo, num programa que manipule datas, no caso de elas terem sido recebidas como *strings*, para transformar cada substring relativa ao dia, ao mês e ao ano em inteiros, para que se possam efetuar operações matemáticas.

g) Convertendo um número numa *string*

No sentido inverso do procedimento *Val*, pode-se transformar uma constante do tipo *integer* ou do tipo *real* numa cadeia de caracteres que pode então ser armazenada numa variável do tipo *string*. Esta ação é conseguida através do procedimento

```
Str(x : integer/real; var s : string);
```

sendo permitido ativá-lo com um argumento da forma y:c:d, onde *c* fixará o comprimento pretendido e *d*, utilizado se y for do tipo *real*, indicará o número de casa decimais.

7.4 Exemplos Parte XII

1. O primeiro exemplo desta seção é uma função para converter uma data dada no formato americano *mm/dd/aa* para o formato brasileiro *dd/mm/aa*. A idéia é extrair a indicação do dia, acompanhado do separador /, inserindo o que foi extraído no início da data. A extração pode ser feita utilizando-se a função *Copy*, seguida da chamada do procedimento *Delete*, e a inserção pode ser realizada com o procedimento *Insert*.

```
{Funcao que converte uma data do formato mm/dd/aaaa para o formato dd/mm/aaaa}
function  ConverteDatas(s : string) : string;
 var Dia : string[3];
 begin
     Dia := Copy(s, 4, 3);
     Delete(s, 4, 3);
     Insert(Dia, s, 1);
     ConverteDatas := s;
 end;
```

2. Naturalmente, seria interessante que antes que esta função fosse ativada houvesse uma verificação da validade da data a ser utilizada como argumento. Abaixo apresentamos uma proposta com esta finalidade.

```
{Funcao que verifica a validade de uma data}
function VerificaData(s : string) : boolean;
var i, d, m, a, r : integer;
     VData : boolean;
     Dia, Mes : string[2];
     Ano : string[4];
begin
    Dia := Copy(s, 1, 2);
    Mes := Copy(s, 4, 2);
    Ano := Copy(s, 7, 4);
    Val(Dia, d, r);
    Val(Mes, m, r);
    Val(Ano, a, r);
    VData := true;
    if (m <= 12) and (m >= 1) and (d <= 31) and (d >= 1)
```

```
        then
            Case m of
                2: if d > 29
                    then
                        Data := false
                    else
                        if ((a mod 4 <> 0) or (a mod 100 <> 0)) and (a mod 400 <> 0)
                            then
                                if d > 28
                                    then
                                        VData := false;
                4, 6, 9, 11: if d > 30
                                then
                                    VData := false;
            end
        else
            VData := false;
    VerificaData := VData;
end;
```

3. A função abaixo inverte uma *string* caractere a caractere, utilizando a indexação das componentes. Para que a *string* retornada possa ser acessada globalmente, é necessário que se atribua a sua componente 0 (zero) uma referência ao seu comprimento que, evidentemente, é igual ao da *string* argumento.

```
function InverteString(var s : string) : string;
var i, c : integer;
    Aux : string;
begin
    c := Length(s);
        for i := 1 to c do
            Aux[i] := s[c – i + 1];
    Aux[0] := Chr(c);
    InverteString := Aux;
end;
```

4. O exercício seis do capítulo cinco solicitava uma função que invertesse um inteiro dado. A utilização das funções *Str* e *Val* permite que se escreva uma função com este objetivo. A idéia é aplicar *Str* transformando o inteiro em uma *string*, inverter esta *string* através da função *InverteString* definida acima e, em seguida, aplicar o procedimento *Val* para obter o número procurado.

```
function InverteNum(n : integer) : longint;
var m, r : integer;
    Aux : string[20];
begin
    Str(n, Aux);
    Aux := InverteStr(Aux);
    Val(Aux, m, r);
    InverteNum := m;
end;
```

Observe que não há necessidade de se verificar se a aplicação de *Val* foi bem sucedida. Neste programa, isto já seria feito pelo próprio sistema, já que *n* foi definida do tipo *integer* e uma entrada incorreta acarretaria um erro de execução. Assim o parâmetro *r* não teve nenhuma aplicação, mas é indispensável.

5. O próximo exemplo apresenta um recurso oferecido pela totalidade dos editores de texto. O objetivo é substituir, em um texto dado, uma cadeia por uma outra cadeia dada (no *Word*, por exemplo, este recurso é ativado através da combinação das teclas <Ctrl> + U). Neste caso, necessitamos localizar a subcadeia (função *Pos*), excluí-la (procedimento *Delete*) e inserir a cadeia dada (procedimento *Insert*).

```
procedure SubstCadeia(var s : string);
var Substituir, SubstituirPor: string;
```

```
    p, c : integer;
begin
    write('Substituir : ');
    readln(Substituir);
    write('Substituir por: ');
    readln(SubstituirPor);
    c := Length(Substituir);
    p := Pos(Substituir, s);
    if p <> 0
        then
            begin
                Delete(s, p, c);
                Insert(SubstituirPor, s, p);
            end
        else
            writeln('O item pesquisado não foi encontrado!');
end;
```

6. Os compiladores da linguagem Pascal ignoram espaços em branco digitados num programa. Uma maneira de se tornar isto possível é, antes da compilação, eliminar todos os espaços em branco "supérfluos", ou seja, deixar duas palavras sempre separadas por um único espaço em branco. A função abaixo realiza tal ação.

```
{Função que exclui espaços em branco desnecessarios}
function ExcluiBrancosSuperfluos(var s : string) : string;
var i, c, NumBrancos : integer;
begin
    i := 1;
    c := Length(s);
    while i < c do
        begin
            NumBrancos := 0;
            while s[i] = ' ' do
                begin
                    NumBrancos := NumBrancos + 1;
                    i := i + 1;
                end;
            if NumBrancos > 1
                then
                    begin
                        i = i - NumBrancos;
                        Delete(s, i, NumBrancos - 1);
                    end;
            i := i + 1;
        end;
end;
```

Observe que a idéia é percorrer toda a cadeia à procura de espaços em branco. Quando um espaço é encontrado, contam-se quantos espaços em branco existem até o próximo caractere e então excluem-se os espaços supérfluos.

7. A questão a seguir é bem interessante. Trata-se de um programa que determine o *dígito verificador* de um número de uma conta corrente, de um número de matrícula de um estudante de uma escola etc. O dígito verificador serve para a prevenção de possíveis erros de digitação. Por exemplo, se a matrícula 30245-7 fosse digitada erroneamente como 39245-7, o erro seria detectado, pois o dígito verificador da conta 39245 seria 6 e não 7. Existem vários métodos para a determinação do dígito verificador. Um deles é dado pelo seguinte algoritmo:

1. Multiplicam-se os números correspondentes aos dígitos da conta, da direita para esquerda, por 2, por 3 etc.

2. Somam-se os produtos obtidos no item 1.

3. Determina-se o resto da divisão da soma obtida no item 2 por 11.

4. Subtrai-se de 11 o resto obtido no item 3

5. Se o valor obtido no item 4 for 10 ou 11, o dígito verificado é igual a zero; senão, o dígito é o valor obtido no item referido.

Por exemplo, se o número da conta for 30245, temos

1. 5x2 = 10, 4x3 = 12, 2x4 = 8, 0x5 = 0, 3x6 = 18

2. 10 + 12 + 8 + 0 + 18 = 48

3. Resto(48, 11) = 4

4. 11 - 4 = 7

5. Dígito verificador = 7.

A função abaixo implementa este algoritmo.

```
{Funcao para determinacao do digito verificador de uma conta}
function DigitoVerificador( s : string) : integer;
type TVetor = array[1..20] of byte;
var i, Comp, Soma, Dv : integer;
    Digitos : TVetor;
        {Procedimento para armazenar os digitos da conta}
        procedure ArmazenaDigitos(var s : string; var d : TVetor; t : integer);
        var  j, r : integer;
        begin
            for j := 1 to t do
                Val(s[j], d[j], r);
        end;
{Funcao DigitoVerificador}
begin
    Comp := Length(s);
    ArmazenaDigitos(s, Digitos, Comp);
    Soma := 0;
    for i := Comp downto 1 do
        Soma := Soma + Digitos[i]*(Comp - i + 2);
    Dv := 11 - Soma mod 11;
    if (Dv = 10) or (Dv = 11)
        then
            Dv := 0;
    DigitoVerificador := Dv;
end;
```

7.5 Exercícios propostos

1. Uma palavra é *palíndromo* se ela não se altera quando lida da direita para esquerda. Por exemplo, *raiar* é *palíndromo*. Escreva um programa que verifique se uma palavra dada é palíndromo.

2. Um dos recursos disponibilizados pelos editores de texto mais modernos é a determinação do número de palavras de um texto. Escreva um programa que determine o número de palavras de uma frase dada.

3. O exercício 16 da seção 6.9 solicitava um programa que convertesse um número dado no sistema decimal para o sistema binário. Pela limitação do sistema em tratar números inteiros, uma solução que tratasse a conversão como sendo do tipo *longint* seria limitada. Escreva uma função para a conversão citada, tratando o valor em binário como uma cadeia de caracteres.

4. Escreva um programa que converta um número do sistema binário, dado como uma cadeia de zeros e uns, para o sistema decimal de numeração.

5. Reescreva a função apresentada no exemplo 7 da seção 7.4 de tal modo que ele possa, além de gerar dígitos verificadores, verificar se uma conta dada (incluindo o dígito verificador) foi digitada incorretamente, incorreção detectada pelo tal dígito verificador.

6. Um algoritmo para a determinação de códigos de barra de produtos é o seguinte:

1. Determina-se a soma dos algarismos das posições ímpares, da esquerda para direita.

2. Calcula-se o triplo da soma dos algarismos das posícões pares, da esquerda para direita.

3. Determina-se a soma dos resultados obttdos em 1 e 2.

4. Determina-se o número que deve ser somado ao resultado obtido em 3 para se obter um múltiplo de dez.

Escreva uma função que implemente o algoritmo acima.

7. As companhias de transportes aéreos costumam representar os nomes dos passageiros no formato *último sobrenome/nome*. Por exemplo, o passageiro Carlos Drumond de Andrade seria indicado por Andrade/Carlos. Escreva um programa que receba um nome e o escreva no formato acima.

8. As normas para a exibição da bibliografia de um artigo científico, de uma monografia, de um livro texto etc., exigem que o nome do autor seja escrito no formato *último sobrenome, seqüência das primeiras letras do nome e dos demais sobrenomes, seguidas de ponto final*. Por exemplo, Antônio Carlos Jobim seria referido por Jobim, A. C.. Escreva um programa que receba um nome e o escreva no formato de bibliografia.

9. É comum que AVISOS, DECLARAÇÕES, CERTIDÕES, etc., tenham seus títulos escritos com as letras separadas por um espaço em branco. Escreva um procedimento que receba uma palavra e a retorne com suas letras separadas por um espaço em branco.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 8 Registros e Arquivos

8.1 Registros (Records)

Um vetor é capaz de armazenar diversos valores, com a restrição de que todos sejam de um mesmo tipo de dado. Um programa que gerencie os recursos humanos de uma empresa manipula dados relativos a cada um dos funcionários que são de tipos diferentes. Por exemplo, para cada funcionário deve-se ter sua matrícula, seu nome, seu endereço, o cargo que ele ocupa, o número de seus dependentes, o seu salário, a data de admissão etc. Observe que nome, matrícula, endereço, data de admissão e cargo que ele ocupa podem ser tratados como *string*. Porém, como, eventualmente, haverá necessidade de se efetuarem operações aritméticas com eles, o número de dependentes deve ser tratado como do tipo *integer* e valor do salário, do tipo *real*.

A utilização de uma variável simples para cada um destes elementos, implicaria, como são vários funcionários, a necessidade de vários vetores, o que poderia atrapalhar a legibilidade e a manutenabilidade do programa, alem de dificultar a possibilidade de armazenamento dos dados em disco, conforme veremos numa seção posterior.

Um *registro* é um conjunto de variáveis, denominadas *campos*, que podem ser de tipos diferentes. Isto é, um *registro* é um *tipo estruturado heterogêneo*. A cada campo de um registro são associados um identificador e um tipo de dado, o que é feito através da seguinte sintaxe:

```
type Identificador = record
    Identificador do campo 1 : tipo de dado;
    Identificador do campo 2 : tipo de dado;
    . . .
    Identificador do campo n : tipo de dado;
end;
```

onde *tipo de dado* pode ser qualquer tipo, simples ou estruturado (inclusive um outro registro).

No programa do comentário feito acima, deveria haver uma definição do seguinte tipo:

```
type TFuncionario = record
    Matr : string[8];
    Nome : string[30];
    Ender : record
        Rua : string[40];
        Num : string[6]
        Bairro : string[20]
        CEP : string[8]
    end;
    Cargo : string[20];
    NumDepend : integer;
    Salario : real
    DataAdm : string[8];
end;
```

O acesso a um campo de um registro é feito através dos identificadores do registro e do campo separados por um ponto. Desta forma, se no programa citado acima declarássemos uma variável *Func* do tipo *TFuncionario*, poderíamos ter as seguintes atribuições:

```
Func.Matr := '1119310'
Func.Nome := 'Vinicius de Moraes';
Func.Ender.Rua := 'Rua Garota de Ipanema';
Func.NumDepend := 8;
```

e comandos do tipo:

```
readln(Func.Matr);
```

```
while Func.Matr <> '0' do
    begin
        readln(Func.Nome);
        readln(Func.Ender.Rua);
        . . .
    end;
```

O comando *with* permite que o identificador do *registro* seja omitido da referência a um dos seus campos, simplificando sobremaneira esta referência. Sua sintaxe é a seguinte :

```
with  Identificador do registro  do
    sequencia de comandos
```

Nos comandos vinculados ao *with*, a referência a um campo do registro é feita apenas com a utilização do identificador daquele campo. Como ocorre em outras situações, se a lista de comandos possuir mais de um comando é necessário se utilizarem os delimitadores *begin* e *end*.

Os comandos de atribuição apresentados acima poderiam ter a seguinte forma:

```
with Func do
    begin
        Matr := '1119310'
        Nome := 'Vinicius de Moraes';
        Ender.Rua := 'Rua Garota de Ipanema';
        NumDepend := 8;
    end;
```

Para exemplificar registros, o programa abaixo recebe as matrículas, os nomes e as notas das avaliações bimestrais de uma relação de alunos e emite um relatório incluindo as matrículas, os nomes e as médias anuais, calculadas como médias aritméticas das avaliações bimestrais. Observe que um dos campos do registro é um vetor, o qual será utilizado para armazenar as notas das avaliações bimestrais de cada aluno. Observe também que toda relação é armazenada num vetor cujas componentes são registros.

```
program ProcessaNotas;
type TVetor1 = array[1..4] of real;
     TRegistro = record
                   Mat : string[9];
                   Nome : string[30];
                   Notas : TVetor1;
                   Media : real;
     end;
     TVetor2 = array[1..60] of TRegistro;
var Aluno : TRegistro;
    Mapa : TVetor2;
    i, j : integer;
    Soma : real;
begin
    i := 1;
    writeln('Digite a matrícula, o nome e as notas (0 (zero) para encerrar)');
    readln(Mapa[i].Mat);
    while Mapa[i].Mat <> '0' do
        with Mapa[i] do
            begin
                readln(Nome);
                Soma := 0;
                for j := 1 to 4 do
                    begin
                        readln(Notas[j]);
                        Soma := Soma + Notas[j];
                    end;
                Media := Soma/4;
```

```
                i := i + 1;
                writeln('Digite   a   matrícula, o  nome e as notas (0 (zero) para encerrar)');
                readln(Mapa[i].Mat);
            end;
    writeln('     Relação das Notas');
    for j := 1 to i - 1 do
        with Mapa[j] do
            writeln(Mat,'     ', Nome,'      ', media:6:2);
end.
```

Evidentemente, o programa acima tem um inconveniente importante! Como os dados são armazenados na memória principal, eles são perdidos quando a máquina é desligada, o que exigiria uma nova digitação destes dados se pretendêssemos uma nova relação das médias. Este inconveniente é solucionado com o uso de *arquivos* que será estudado a seguir.

8.2 O que são arquivos

Até o momento, os dados manipulados pelos nossos programas (dados de entrada, dados gerados pelo programa e resultados do processamento) foram armazenados na memória do computador que, como já foi dito, é uma memória volátil no sentido de que todos as informações nela armazenadas são perdidas quando a execução do programa é encerrada.

É evidente que um programa que gerencia os recursos humanos de uma empresa não pode manipular os dados relativos aos funcionários apenas na memória do computador. Isto implicaria, por exemplo, a necessidade de que fossem digitados todos os dados em todas as execuções do programa. É evidente, portanto, que os dados relativos a cada um dos funcionários da empresa devem estar armazenados, de forma permanente, em um disco, de modo que o programa que gerencia os recursos humanos possa acessá-los em execuções distintas.

Em computação, dados e informações reunidos e armazenadas num disco constituem um *arquivo* e a linguagem Pascal, naturalmente, permite que se manipulem arquivos em discos, fornecendo recursos para a realização das operações básicas que podem ser neles executadas: criação de um arquivo, alteração dos dados de um arquivo, exclusão de dados de uma arquivo, inclusão de novos dados no arquivo, exibição (na tela ou em formato impresso) do conteúdo de uma arquivo etc.

Estamos, até agora, utilizando o termo *armazenar* para indicar o ato de "escrever" um dado num arquivo. Estamos fazendo isto procurando ser coerente com a utilização do termo em relação à memória do computador. No caso de armazenamento num arquivo é mais comum se utilizar o verbo *gravar*.

8.3 Arquivos de registros

Os arquivos de uso mais comum na prática de programação em Pascal são os arquivos que armazenam dados oriundos de registros. Por exemplo, um sistema que gerencie uma locadora de fitas deve manipular um arquivo que armazene, para cada fita, um código, o título do filme, o tema, a data de aquisição, o preço de custo, o valor da locação etc. No momento da entrada estes dados podem ser armazenados num registro para serem, em seguida, armazenados num arquivo. É comum se dizer que o conjunto dos campos dos registros do arquivo define a *estrutura* do arquivo.

Uma grande parte das operações que são feitas em arquivos requer a verificação de que o registro que se pretende manipular está armazenado no arquivo. Isto exige que os registros possuam um campo cujos valores sejam distintos entre si, sendo o valor deste campo um identificador do referido registro. Esta é a função de campos do tipo CPF, matrículas, placas de carro etc. Um campo identificador dos registros de um arquivo é chamado *chave*.

8.3.1 Definido um tipo arquivo

Para que seja utilizado como tipo de parâmetro de subprogramas e para que seja possível a interação entre variáveis definidas no programa e arquivos gravados em disco, é interessante que se defina um *tipo arquivo*, o que é feito através da seguinte sintaxe:

type Identificador : **file of** tipo de dado;

onde *tipo de dado* pode ser qualquer tipo.

Por exemplo, um programa que gerencie o estoque dos produtos de uma loja poderia ter definições e declarações do seguinte tipo:

```
type   TRegistro = record
            Cod : string[8]
            Nome : string[30];
            Estoque: integer;
            PrecoCusto : real;
            Priorid : boolean;
         end;
         TArquivo = file of TRegistro;
var   Produto : TRegistro;
      Arq : TArquivo.
```

Também podemos declarar diretamente variáveis do tipo *file* como mostra o seguinte exemplo:

```
var Serie : file of integer;
    Texto : file of char;
```

Com estas declarações, a variável *Serie* pode ser associada a um arquivo que pode armazenar uma série de inteiros e a variável *Texto* pode ser associada a um arquivo capaz de armazenar um texto. Em seguida, veremos como é feita a associação entre uma variável de um programa e um arquivo gravado em um disco. No caso de um arquivo do tipo *file of char*, o sistema disponibiliza um tipo específico o qual será estudado posteriormente.

8.3.2 Associando variáveis a arquivos

É fácil compreender que a manipulação por um programa de um arquivo que possa ser gravado (ou possa já estar gravado) num disco exige dois identificadores: um que identifica o arquivo no disco e um outro que, como uma variável, o identificará no programa. Assim, há a necessidade de uma associação entre variáveis a serem utilizadas no programa e arquivos gravados (ou a serem gravados) em disco. Esta associação é feita através do procedimento *Assign*, que possui a seguinte sintaxe:

Assign(**var** f : TArquivo; NomeArquivo : **string**);

Aí (e nas referências futuras) *TArquivo* é um tipo *file* definido previamente e a execução deste procedimento associa o nome do arquivo gravado no disco (o qual deve ser passado para o parâmetro *NomeArquivo*, podendo incluir o "caminho" da *pasta* que conterá o arquivo) com uma variável do tipo *TArquivo*, que será utilizada no programa para se fazer referência ao arquivo que se está manipulando. Depois da execução deste procedimento, qualquer referência no programa à variável passada para o parâmetro *f* se refletirá no arquivo cujo nome foi passado para o parâmetro *NomeArquivo*. Este procedimento é indispensável para a manipulação de um arquivo e precede toda e qualquer referência a ele. Observe que toda passagem de parâmetro do tipo *file* tem que ser feita por referência.

8.3.3 Criando um arquivo

A criação de um arquivo em disco é feita através do procedimento

Rewrite(**var** f : TArquivo);

que deve ser ativado após um procedimento *Assign*. A sua execução redundará na criação (no disco da unidade em uso) de um arquivo com o nome passado para o segundo argumento do procedimento *Assign*.

Por exemplo, a execução do programa

```
program GeraArquivo;
type   TRegistro = record
                Matr :  string[9];
                Nome : string[40];
                SalarioBruto, SalarioLiquido, Inss : real;
                NumDepend : integer;
            end;
            TArquivo = file of TRegistro;
var Arq : TArquivo;
begin
    Assign(Arq, 'A:Folha.arq');
    Rewrite(Arq);
end.
```

gera, no disco colocado na unidade A, um arquivo denominado *Folha.arq* com tamanho zero byte, já que nada nele foi armazenado.

É necessário muito cuidado com a utilização do procedimento *Rewrite* pois sua ativação com argumento associado a um arquivo que já existe provoca a perda de todos os registros do arquivo. Um cuidado possível será mostrado a seguir.

8.3.4 Gravando dados num arquivo

No exemplo anterior, um arquivo foi criado, mas nenhum dado foi nele armazenado. A gravação de dados em um arquivo é feita através do comando

write(**var** f : TArquivo; v : tipo de dado);

onde *tipo de dado* é o tipo de dado associado ao tipo *TArquivo*. Quando da sua execução, os dados passados para o parâmetro *v* são gravados no arquivo associado à variável passada para o parâmetro *f*.

Por exemplo, o programa abaixo gera um arquivo no disco da unidade A capaz de armazenar inteiros e nele grava os quadrados dos 50 primeiros números inteiros.

```
program GravaQudrados;
type TArquivo = file of integer;
var Arq : TArquivo;
      i, Quad : integer;
begin
    Assign(Arq, 'A:Quadrados');
    Rewrite(Arq);
    for i := 1 to 50 do
        begin
            Quad := i*i;
            write(Arq, Quad);
        end;
end.
```

Após a execução deste programa, a exibição do conteúdo do disco contido na unidade A vai mostrar a existência de um arquivo denominado *Quadrado* com um tamanho de 100 Bytes. A adoção da denominação *Quadrado* e não *Quadrados*, como estabelecido no comando *Assign*, se dá pelo fato de que a interação do Pascal com discos é feita pelo sistema operacional DOS, que não aceita um identificador de arquivo com mais de oito caracteres.

Para um outro exemplo, poderíamos reescrever o programa *GeraArquivo* acima, para que, logo após a criação do arquivo *Folha.arq*, os dados pretendidos fossem armazenados:

```
{Programa para gerar um arquivo e gravar dados no arquivo gerado}
program GeraArquivo;
type     TRegistro = record
                Matr :  string[9];
```

```
                Nome : string[40];
                SalarioBruto, SalarioLquido, Inss : real;
                NumDepend : integer;
            end;
            TArquivo = file of TRegistro;
    var Arq : TArquivo;
        {Procedimento para gravar dados num arquivo}
        procedure GravaDados(var f : TArquivo);
        var r : TRegistro;
        begin
            with r do
                repeat
                    writeln('Digite matricula, nome, salario e numero de dependentes (0 para encerrar)');
                    readln(Matr);
                    if Matr <> '0'
                        then
                            begin
                                readln(Nome, SalarioBruto, NumDepend);
                                write(f, r);
                            end;
                until Matr = '0';
            end;
    {Programa principal}
    begin
        Assign(Arq, 'A:Folha.arq');
        Rewrite(Arq);
        GravaDados(Arq);
    end.
```

É importante observar que os registros são gravados seqüencialmente e o sistema indexa cada um deles com os índices 0, 1, 2, ..., de acordo com a ordem em que eles foram gravados.

Vale observar, também, que o procedimento *write* é o mesmo que vimos usando para exibir a saída dos programas no vídeo. Isto sempre acontece quando o parâmetro *f* é omitido. Um outro detalhe importante é que o sistema contém um arquivo *Lst* que gerencia o relacionamento do sistema com a impressora conectada ao computador. Quando o arquivo *Lst* é passado como argumento para o comando *write*, o conteúdo do argumento passado para o segundo parâmetro é impresso. Este arquivo pertence à unidade *Printer* e a sua utilização exige que esta unidade seja relacionada na lista das unidades a serem utilizadas pelo programa.

Por exemplo, a execução do programa

```
    {Programa para gerar uma tabela de conversão de temperaturas em graus fahrenheit para graus
Celsius}
    program TabelaDeConversaoFarenheitCelsius;
    uses Crt;
    var Farenheit : integer;
         Celsius: real;
    begin
        writeln(Lst, 'Tabela de conversao de temperaturas');
        writeln(Lst, '------------------------------------------');
        writeln(Lst, '          Farenheit        |          Celsius');
        writeln(Lst, '------------------------------------------');
        for Farenheit := 20 to 80 do
            begin
                Celsius := 5.0*(Farenheit - 32)/9;
                writeln(Lst, '          ', Farenheit,'      ', Celsius:0:2);
            end;
    end.
```

imprimiria a tabela da página 72.

8.3.5 Abrindo e fechando um arquivo

A realização de operações num arquivo exige que ele esteja disponibilizado para tal. O ato de disponibilizar um arquivo para nele realizar operações é chamado *abertura* do arquivo, que em Pascal é feito através do procedimento

```
Reset(var f : TArquivo);
```

Naturalmente, o arquivo a ser aberto é aquele cujo nome foi associado ao parâmetro *f* por um procedimento *Assign*, cuja ativação deve preceder a ativação do comando. Naturalmente, também, a ativação de um procedimento *Reset* para um arquivo que não existe provoca um erro de execução.

É prudente então se preceder a ativação de um procedimento *Reset* de uma função que verifique a existência do arquivo que se pretende abrir. Isto pode ser feito com uma função como a que apresentamos a seguir.

```
function ExisteArquivo(var f : TArquivo) :  boolean;
begin
     {$I-}
     Reset(f);
     if IOResult = 0
          then ExisteArq := true
          else ExisteArq := false;
     {$I+}
end;
```

A expressão {$I-} é uma *diretiva de compilação* e o objetivo de uma instrução como esta é definir como a compilação do programa deve ser realizada. A diretiva {$I-} inibe a verificação de erros de entrada e saída que aconteceria se o arquivo associado à variável passada para *f* não existisse. A função *IOResult* é uma função predefinida que retorna 0 (zero) se a ultima operação de entrada/saída foi realizada com sucesso. A função é encerrada com {$I+} para reativar a verificação de erros de entrada e saída pelo sistema.

A função *ExisteArq* pode ser utilizada quando se pretende criar um arquivo novo, para evitar a destruição de um arquivo já existente, conforme foi discutido na apresentação do comando *Rewrite*. Desta forma seria interessante que o programa *GeraArq* discutido acima fosse modificado para a seguinte versão:

```
{Programa para gerar um arquivo}
program GeraArquivo;
type     TRegistro = record
               Matr :  string[9];
               Nome : string[40];
               SalarioBruto, SalarioLquido, Inss : real;
               NumDepend : integer;
          end;
          TArquivo = file of TRegistro;
var Arq : TArquivo;
          {Funcao para verificar a existencia de um arquivo}
          function ExisteArquivo(var f : TArquivo): boolean;
          begin
               {$I-}
               Reset(f);
               if IOResult = 0
                     then ExisteArquivo := true
                     else ExisteArquivo := false;
               {$I+}
          end;
          {Procedimento para gravar dados num arquivo}
          procedure GravaDados(var f : TArquivo);
          var r : TRegistro;
```

```
        begin
            with r do
                repeat
                    writeln('Digite matricula, nome, salario e numero de dependentes (0 encerra)');
                    readln(Matr);
                    if Matr <> '0'
                        then
                            begin
                                readln(Nome, SalarioBruto, NumDepend);
                                write(f, r);
                            end;
                until Matr = '0';
        end;
    {Programa principal}
    begin
        Assign(Arq, 'A:Folha.arq');
        if not ExisteArquivo(Arq)
            then
                begin
                    Rewrite(Arq);
                    GravaDados(Arq);
                end
            else
                writeln('Arquivo A:Folha.arq  ja existe');
    end.
```

Embora ainda não tenhamos feito, é necessário que todo arquivo criado por um procedimento *Rewrite* ou aberto por um procedimento *Reset* seja *fechado* para que o sistema operacional DOS realize as atualizações necessárias, inclusive na tabela de diretórios. Esta ação é obtida com o procedimento

Close(**var** f : TArquivo);

que deve ser ativado em relação a todos os arquivos abertos ou criados pelo programa.

8.3.6 Exibindo o conteúdo de um arquivo

A exibição no vídeo ou a impressão do conteúdo de um arquivo requer que cada registro seja armazenado na memória do computador para, em seguida, ser exibido ou impresso por um comando *writeln*, sem argumento do tipo *Tarquivo* para exibição no vídeo, ou com o argumento *Lst* para impressão. O armazenamento na memória de um registro gravado num arquivo é feito através do procedimento

read(**var** f : **TArquivo**; v : tipo de dado);

onde *tipo de dado* é o tipo do registro do arquivo. A execução deste procedimento redunda na "leitura" dos dados do registro disponibilizado para tal e o conseqüente armazenamento destes dados na variável *v*.

Quando se abre um arquivo com o procedimento *Reset*, o registro disponível para leitura é o primeiro registro que foi gravado no arquivo, ou seja, o registro de índice zero. O registro disponibilizado para leitura (e para gravação) é chamado *registro corrente* e dizemos que o *ponteiro de leitura e gravação aponta* para o tal registro. Após a execução de um procedimento *read*, o ponteiro de leitura e gravação avança para o registro seguinte. O sistema possui a função

Eof (**var** f : TArquivo) : **boolean**;

que retorna *true* quando o final do arquivo é atingido. Isto acontece após a leitura do último registro gravado, após a execução de um procedimento *Rewrite* e após a execução de um comando *Reset* em um arquivo onde dados não foram gravados.

A exibição do conteúdo de um arquivo pode ser feita abrindo-se o arquivo por um *Reset* e, em seguida, percorrendo-o até o seu final e combinado-se os comandos *read* e *writeln*:

```
{Procedimento para exibir o conteudo de um arquivo}
procedure ExibeArquivo(var f : TArquivo);
var r : TRegistro;
begin
    Reset(f);
    while not Eof(f) do
        begin
            read(f, r);
            with r do
                writeln(Matr:20, Nome:20, SalarioBruto:6:2, NumDepend:3);
        end;
end;
```

8.3.7 Localizando um registro num arquivo

Uma operação muito comum em arquivos é a verificação de que um determinado registro está armazenado no arquivo. Esta operação é normalmente chamada *consulta*, *pesquisa* ou *busca* e deve ser feita de acordo com o valor da chave do registro ou de um outro campo que, relativamente, identifique o registro. No exemplo que estamos discutindo, a consulta pode ser feita pelo campo *Matr* (de matrícula) ou pelo campo *Nome*. Em geral, a consulta se processa com a localização do registro, a conseqüente exibição do seu conteúdo e o retorno da posição que ele ocupa no arquivo.

A localização do registro pode ser feita, abrindo o arquivo com *Reset* e o percorrendo até que o valor da chave seja encontrado; a exibição do seu conteúdo pode ser feita através dos procedimentos *read* e *writeln* e a posição que ele ocupa no arquivo é fornecida pelo procedimento

FilePos(**var** f : arquivo) : **integer**;

que retorna o índice do registro corrente.

A função abaixo verifica se um registro dado pelo campo *Matr* está gravado no arquivo *Folha.arq* que estamos utilizando como exemplo. Em caso positivo, a função exibe o conteúdo do arquivo e retorna o índice do registro respectivo; em caso negativo, a função emite mensagem indicando o insucesso da busca e retorna o valor –1.

```
{Funcao que realiza consulta por matricula num arquivo}
function Consulta(var f : TArquivo; Mat : string; var r : TRegistro) : integer;
begin
    Reset(f);
    read(f, r);
    while (not Eof(f)) and (r.Matr <> Mat) do
        read(f, r);
    if r.Matr = Mat
        then
            begin
                with r do
                    writeln(Matr:6,  Nome:20, SalarioBruto:6:2, NumDepend:3);
                Consulta := FilePos(f);
            end
        else
            begin
                writeln('Matricula nao cadastrada');
                Consulta := -1;
            end;
end;
```

Analisando estritamente esta função, não haveria necessidade de que *r* fosse um parâmetro, podendo ser apenas uma variável local. Ele está sendo definido como um parâmetro por referência para utilizações em outros procedimentos como veremos a seguir.

Outra observação importante é que o método utilizado para realização da pesquisa pode ser bastante ineficiente nos casos em que o registro procurado for um dos últimos do arquivo. No capítulo seguinte, apresentaremos um método mais eficiente quando os registros estão ordenados pelo campo chave.

8.3.8 Alterando o conteúdo de um registro

Às vezes há necessidade de que os dados de um registro sejam alterados. No arquivo que estamos utilizando como exemplo, isto poderia ocorrer no caso de uma promoção de um funcionário que implicasse um aumento no seu salário bruto, no caso de uma funcionária que alterou o seu nome em função de um casamento ou no caso de um funcionário que teve seu número de dependentes aumentado.

Um procedimento para alterar os dados de um registro deve receber a variável do tipo *TArquivo* associado ao arquivo que se pretende alterar e o valor do campo chave para indicar qual é o registro no qual se pretende fazer a alteração. Este procedimento deve abrir o arquivo com um procedimento *Reset*, localizar o registro com a função *Consulta* definida acima para obter a posição do registro pretendido e, com a informação desta posição, deve-se posicionar o ponteiro de leitura e gravação naquele registro e realizar as alterações que são desejadas. Para posicionar o ponteiro de leitura e gravação num determinado registro, utiliza-se o procedimento

Seek(**var** f : TArquivo; n : **integer**);

que posiciona o ponteiro de leitura e gravação no registro de índice n.

Para exemplificar, o procedimento abaixo altera o campo *Nome* do registro do arquivo *A:Folha.arq*, cujo campo *Matr* for dado.

```
{Procedimento que altera um nome de um registro de matricula dada}
procedure AlteraRegistro(var f : TArquivo; Mat : string; var r : TRegistro);
var n : integer;
     Sim : char;
begin
    Reset(f);
    n := Consulta(f, Mat, r);
    if n <> -1
        then
            begin
                writeln('Deseja alterar o campo nome)
                readln(Sim);
                if UpCase(Sim) = 'S'
                    then
                        begin
                            writeln('Digite o novo nome');
                            readln(r.Nome);
                            Seek(f, n);
                            write(f, r);
                        end;
            end;
end;
```

8.3.9 Incluindo novos registros num arquivo

A inclusão de um novo registro no final de um arquivo é bastante simples, bastando posicionar o ponteiro de leitura e gravação no final do arquivo e, em seguida, gravar o conteúdo da variável que recebeu os dados no registro. Para se posicionar o ponteiro de leitura e gravação no final do arquivo, pode-se utilizar a função

```
FileSize(var f : TArquivo) : integer;
```

que retorna o número de registros do arquivo associado a *f*, às vezes chamado *tamanho do arquivo*. Com esta função, a ativação do procedimento

```
Seek(f, FileSize(f))
```

posicionará o ponteiro de leitura e gravação no final do arquivo.

```
procedure IncluiRegistro(var f : TArquivo; var r : TRegistro);
var n : integer;
    Reg : TRegistro;
begin
    n := Consulta(f, r.Matr, Reg);
    if n = -1
        then
            begin
                Seek(f, FileSize(f));
                write(f, r);
            end
        else
            writeln('Matricula ja cadastrada');
end;
```

8.3.10 Excluindo (fisicamente) um registro de um arquivo

Outra operação muito utilizada em arquivos é a exclusão de um registro. No exemplo que estamos utilizando, esta operação seria necessária na ocasião de um pedido de demissão de um funcionário. Uma possível solução é, após localizar o registro, gravar todos os outros registros num arquivo temporário, *Temp*, excluir do disco o arquivo original e renomear o arquivo *Temp* com o nome do arquivo original. A exclusão de um arquivo é feita com o procedimento

```
Erase(var f : TArquivo);
```

e a renomeação com o procedimento

```
Rename(var f : TArquivo; NovoNome : string);
```

Estes dois procedimentos não devem ser ativados com arquivos abertos e, portanto, suas execuções devem ser precedidas de chamadas ao procedimento *Close*.

```
procedure ExcluiRegistro(var f : TArquivo; Mat : string);
var n, i : integer;
    Reg : TRegistro;
    Sim : char;
    Temp : TArquivo;
begin
    n := Consulta(f, Mat, Reg);
    if n <> -1
        then
            begin
                write('Deseja excluir o registro (S/N)?');
                readln(Sim);
                if UpCase(Sim) = 'S'
                    then
                        begin
                            Assign(Temp, 'A:Temp');
                            Rewrite(Temp);
                            Seek(f, 0);
                            i := 0;
```

```
                        while not Eof(f) do
                            if i <> n
                                then
                                    begin
                                        read(f, Reg);
                                        write(Temp, Reg);
                                        i := i + 1;
                                    end;
                        Close(f);
                        Close(Temp);
                        Erase(f);
                        Rename(Temp, 'A:Folha.arq');
                    end;
            end
                else
                    writeln('Registro nao encontrado');
end;
```

Naturalmente, a utilização do arquivo temporário poderá acarretar uma queda na performance do programa. O procedimento abaixo, talvez de compreensão menos simples, realiza exclusões de registros sem a utilização de arquivo temporário. A idéia é gravar, a partir do registro seguinte ao registro a ser excluído, os dados do registro corrente no registro anterior. O único problema é que o último registro fica duplicado. Porém, ele pode ser excluído pelo procedimento

Truncate(**var** f : TArquivo);

que exclui todos os registros a partir do registro corrente.

```
procedure ExcluiRegistro(var f : TArquivo; Mat : string);
var n, i : integer;
    Reg : TRegistro;
    Sim : char;
begin
    n := Consulta(f, Mat, Reg);
    if n <> -1
        then
            begin
                write('Deseja excluir o registro (S/N)?');
                readln(Sim);
                if UpCase(Sim) = 'S'
                    then
                        begin
                            Seek(f, n + 1);
                            while not Eof(f) do
                                begin
                                    read(f, Reg);
                                    Seek(f, FilePos(f) - 2);
                                    write(f, Reg);
                                    Seek(f, FilePos(f) + 1);
                                end;
                            Seek(f, FileSize(f) - 1);
                            Truncate(f);
                        end;
            end;
    Close(f);
end;
```

Deve ser observada a necessidade das diversas chamadas ao procedimento *Seek* para posicionamento do ponteiro de leitura e gravação.

8.3.11 Excluindo (logicamente) um registro de um arquivo

A exclusão de um registro discutida acima pode ser classifcada como *exclusão física* pelo fato de que todo o registro é, de fato, deletado do arquivo, não permanecendo no disco nenhuma referência a nenhum de seus dados. Este tipo de exclusão pode, mesmo com a confirmação do desejo de fazê-lo, implicar em exclusões indevidas, além de não permitir uma recuperação posterior devido a algum tipo de arrependimento. Pode-se munir um arquivo de mecanismo que permita as chamadas *exclusões lógicas*, que elimina o registro para todas as operações no arquivo, mas não o exclui do mesmo.

Para se ter exclusões lógicas num arquivo, é necessário se definir um campo, digamos *Marca*, do tipo, digamos, *char*, inicializado com, digamos, espaço em branco e que receberá um caractere definido pelo programador (digamos *****) quando se pretende a exclusão lógica do registro. Desta forma, exibições de registros e consultas só levariam em conta os registros para os quais o campo *Marca* fosse diferente de ***.** Teríamos então as seguintes modificações nos procedimentos e funções já estudadas:

```
type    TRegistro = record
                Marca : char
                Matr :  string[9];
                Nome : string[40];
                SalarioBruto, SalarioLquido, Inss : real;
                NumDepend : integer;
            end;
            TArquivo = file of TRegistro;
var Arq : TArquivo;
            {Procedimento para gravar dados num arquivo}
            procedure GravaDados(var f : TArquivo);
            var r : TRegistro;
            begin
                with r do
                    repeat
                        writeln('Digite matricula, nome, salario e numero de dependentes (matricula 0 
para encerrar)');
                        readln(Matr);
                        if Matr <> '0'
                            then
                                begin
                                    Marca := ‘ ‘;
                                    readln(Nome, SalarioBruto, NumDepend);
                                    write(f, r);
                                end;
                    until Matr = '0';
            end;

            {Procedimento para exibir o conteudo de um arquivo}
            procedure ExibeArquivo(var f : TArquivo);
            var r : TRegistro;
            begin
                Reset(f);
                while not Eof(f) do
                    begin
                        read(f, r);
                        if r.Marca <> ‘*’
                            then
                                with r do
                                    writeln(Matr:20, Nome:20, SalarioBruto:6:2, NumDepend:3);
                    end;
            end;
```

```
{Funcao que realiza consulta por matricula num arquivo}
function Consulta(var f : TArquivo; Mat : string; var r : TRegistro) : integer;
begin
    Reset(f);
    read(f, r);
    while (not Eof(f)) and (r.Matr <> Mat) do
        read(f, r);
    if (r.Matr = Mat) and (r.Marca <> '*')
        then
            begin
                with r do
                    writeln(Matr,  Nome, SalarioBruto, NumDepend);
                Consulta := FilePos(f);
            end
        else
            begin
                writeln('Matricula nao cadastrada');
                Consulta := -1;
            end;
end;
```

```
{Procedimento para excluir logicamente um registro}
procedure ExcluiLogicamenteUmRegistro(var f : TArquivo; Mat : string);
var n, i : integer;
    Reg : TRegistro;
    Sim : char;
begin
    n := Consulta(f, Mat, Reg);
    if n <> -1
        then
            begin
                write('Deseja excluir o registro (S/N)?');
                readln(Sim);
                if UpCase(Sim) = 'S'
                    then
                        begin
                            Seek(f, n);
                            read(f, Reg);
                            Reg.Marca := '*';
                            Seek(f, n);
                            write(f, Reg);
                        end;
            end;
    Close(f);
end;
```

8.4 Arquivo texto

8.4.1 O que são arquivos texto

Outro tipo de arquivo que a linguagem Pascal oferece é capaz de armazenar cadeias de caracteres, podendo então ser utilizado para armazenar textos. Para isto é oferecido um arquivo do tipo *text* que, sendo um tipo predefinido, pode ser utilizado para se declarar variáveis e ser usado como tipo de parâmetros de procedimentos e de funções.

Um arquivo do tipo *text* pode ser considerado uma seqüência de caracteres dispostos em linha, sendo que cada linha é encerrada por uma marca de fim de linha, definida pelo caractere, cujo código ASCII decimal é

igual a 13 (treze) (no teclado este caractere é obtido através da tecla <Enter>).

Os procedimentos *Rewrite*, *Reset*, *Close*, *Erase* e *Rename* e a função *Eof* podem ser ativados com argumentos do tipo *text*, resultando em ações idênticas àquelas que ocorrem quando os argumentos são de outros tipos de arquivos.

Os comandos *read* e *write* têm uma forma especial quando o argumento é do tipo *text*. É possível ativá-los passando mais de uma variável como argumentos e ainda variáveis de tipos diferentes do tipo *char*. Neste caso, o próprio sistema transforma o dado no formato do tipo da variável e, então, o armazena. Por exemplo, se *t* é uma arquivo do tipo *text* e *i* é uma variável inteira, a ativação do procedimento

```
read(t, i)
```

lerá do arquivo associado a *t* uma seqüência de dígitos, transformando-a em um número inteiro e o armazenará em *i*. Naturalmente, se a seqüência não puder ser transformada num inteiro, ocorrerá um erro de execução e a execução do programa é abortada.

Em adição ao comando *read*, o sistema possui o comando *readln*, que, após ler um ou mais dados do arquivo e os armazenar em uma ou mais variáveis, posiciona o ponteiro de leitura na próxima marca de fim de linha. Do mesmo modo, existe o comando *writeln* que, após gravar um ou mais dados no arquivo, grava uma marca de fim de linha.

8.4.2 Gravando um texto

A gravação de um texto num arquivo pode ser feita dando entrada no texto caractere a caractere através da função *ReadKey* e gravando cada caractere digitado com o procedimento *write*.

```
procedure GravaTexto( var t : text);
var c : char;
     i : integer;
begin
   writeln('Digite o texto (<Esc> para encerrar)');
   c := ReadKey;
   i := 1;
   while c <> Chr(27) do    {Chr(27) = <Esc>}
      begin
         write(t, c);
         if c = Chr(13)
            then
               begin
                  writeln;
                  i := 0 ;
               end
            else
               write(c);
         c := ReadKey;
         i := i + 1;
         if i = 80     {Número de colunas do vídeo}
            then
               c := Chr(13);
      end;
end;
```

Observe que, como é útil que o texto apareça na tela à medida que está sendo digitado, há a necessidade do comando *write*(c). Este comando é substituído por um comando *writeln* (para que ocorra a mudança de linha), toda vez que a tecla <*Enter*> é digitada, quando se pretende um parágrafo, ou quando da atribuição *c* := Chr(13). Esta atribuição, que ocorre quando *i* = 80, é para garantir a mudança de linha ao final da tela e para que seja gravada uma marca de fim de linha. Esta marca de fim de linha é importante quando da leitura do arquivo através do comando *readln*.

8.4.3 Exibindo e ampliando o conteúdo de um arquivo texto

A utilização de arquivos texto permite que se desenvolva um programa que exiba o conteúdo de qualquer arquivo gravado em ASCII. Para isto consideramos uma variável do tipo *string s* e, através de *readln*, armazenamos cada linha do arquivo texto em *s* e a exibimos através de *writeln*.

```
program ExibeTexto;
uses Crt;
 var Texto : text;
      s, NomeArq : string;
         {Procedimento para exibicao de um arquivo texto}
         procedure ExibeArquivoTexto(var t : text);
         begin
             Reset(t);
             while not Eof(t) do
                   begin
                      readln(t, s);
                      writeln(s);
                   end;
             Close(t);
          end;
{programa principal}
begin
     writeln('Digite o nome do arquivo ');
     readln(NomeArq);
     Assign(Texto, NomeArq);
     ExibeArquivoTexto(Texto);
end.
```

O conteúdo de um arquivo texto pode ser ampliado com a utilização do procedimento

```
Append(var t : text);
```

que abre um arquivo texto e posiciona o ponteiro de leitura e gravação no final do arquivo, permitindo então que o arquivo seja ampliado:

```
program AmpliaTexto;
uses Crt;
var Texto : text;
      s, NomeArquivo : string;
         {Procedimento para gravar um texto, ampliando um arquivo}
         procedure GravaTexto( var t : text);
         var c : char;
                i : integer;
         begin
             Append(t);
             writeln('Digite o texto (<Esc> para encerrar)');
             c := ReadKey;
             i := 1;
             while c <> Chr(27) do
                  begin
                      write(t, c);
                      if c = Chr(13)
                           then
                                begin
                                     writeln;
                                     i := 0 ;
                                end
```

```
                    else
                        write(c);
                    c := ReadKey;
                    i := i + 1;
                    if i = 80     {Numero de colunas do video}
                        then
                            c := Chr(13);
                end;
            Close(t);
        end;
        {Funcao que verifica a existencia de um arquivo}
        function ExisteArquivo(var f : text): boolean;
        begin
            {$I-}
            Reset(f);
            if IOResult = 0
            then ExisteArquivo := true
            else ExisteArquivo := false;
            {$I+}
        end;
{programa principal}
begin
    writeln('Digite o nome do arquivo ');
    readln(NomeArquivo);
    Assign(Texto, NomeArquivo);
    If ExisteArquivo(Texto)
        then
            GravaTexto(Texto)
        else
            writeln('Arquivo ', NomeArquivo, ' não existe');
end.
```

8.5 Exercícios propostos

1. Escreva um programa que, através de um menu de opções, utilize as funções e os procedimentos estudados neste capítulo e que, portanto, seja capaz de

a) criar e gravar dados num arquivo de registros;

b) exibir o conteúdo de um arquivo;

c) alterar dados de um registro de um arquivo;

d) incluir novos registros em um arquivo;

e) excluir um registro de um arquivo.

2. Escreva um programa que reúna dois arquivos de registros de mesma estrutura *Mat : string[10]* e *Nome : string[40]* num terceiro arquivo.

3. Escreva um programa que, dado um arquivo cujos registros possuem os campos *Nome* (do tipo *string*) e *Salario* (do tipo *real*), gere um arquivo dos registros cujo campo *Salario* é maior que 5 000.

4. Escreva duas versões de um procedimento que inclua um registro dado num arquivo de registros ordenado por um campo *Mat*, de modo que o arquivo se mantenha ordenado por este campo. As duas versões devem diferir pela utilização ou não de um arquivo auxiliar.

5. Escreva um programa que, dados dois arquivos ordenados por um campo *Mat*, gere um terceiro arquivo também ordenado.

6. Escreva um programa que dados dois arquivos cujos registros têm os campos *Cpf : string[12]* e *Nome : string[40]* gere um arquivo contendo os registros comuns aos dois arquivos.

7. Escreva um procedimento que troque as posições de dois registros de um arquivo.

8. Escreva um programa que exclua os comentários de um programa escrito em Pascal.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 9 Pesquisa e Ordenação

9.1 Pesquisa

Neste capítulo, discutiremos dois problemas clássicos de computação. O primeiro deles, chamado *pesquisa*, *busca* ou *consulta*, consiste em se verificar se um dado valor está armazenado num vetor ou num campo de um registro de um arquivo.

São vários os exemplos de pesquisas em computação. Uma busca por páginas da *internet* que contenham um determinado assunto; uma busca no Registro Nacional de Veículos Automotores (RENAVAM) na tentativa de se encontrar o nome do proprietário do veículo de uma placa dada; uma busca nos registros da Receita Federal a respeito de um CPF dado.

Como os registros de um arquivo podem ser armazenados num vetor, mostraremos as soluções dos dois problemas aplicados a vetores.

9.1.1 Pesquisa seqüencial

O método de busca mais simples de compreender é o algoritmo que temos utilizado até agora em exemplos e em respostas de exercícios propostos. Este método é chamado *pesquisa seqüencial* e consiste em se percorrer o vetor comparando-se o valor de cada componente com o valor pesquisado. A pesquisa se encerra quando o valor pesquisado é encontrado, ou quando se atinge o final do vetor, significando, neste caso, que a pesquisa não foi bem sucedida.

A função abaixo pesquisa num vetor um valor passado para o parâmetro *x* e retorna a posição ocupada por *x*, se a pesquisa foi bem sucedida, ou o valor –1, no caso contrário.

```
{Funcao que verifica se um valor dado e componente de um vetor}
function PesquisaSequencial(var v : TVetor; t : integer; x : real) : integer;
var j : integer;
begin
    PesquisaSequencial := -1;
    j := 1;
    while (v[j] <> x) and (j < t) do
        j := j + 1;
    if v[j] = x
        then
            PesquisaSequencial := j;
end;
```

9.1.2 Pesquisa binária

É muito fácil perceber que o método da pesquisa binária é bastante ineficiente: imagine que este método fosse utilizado para se pesquisar a palavra *zurzir* num dicionário da língua portuguesa (a propósito, *zurzir* significa açoitar ou criticar com extrema violência).

Quando a relação está ordenada, existe um método de busca, chamado *pesquisa binária*, bem mais eficiente do que a pesquisa seqüencial: compara-se o elemento pesquisado com a componente "central" da relação; se forem iguais, a pesquisa é encerrada com sucesso; se o elemento pesquisado for menor que a componente central repete-se a pesquisa em relação à "primeira metade" da relação; se o elemento pesquisado for maior repete-se a pesquisa em relação à "segunda metade" da relação. Por exemplo, uma pesquisa do número 7 na relação {1, 3, 4, 5, 6, 8, 10, 11, 12, 15, 18, 19, 20, 21, 22, 25, 26} começaria comparando-se 7 com 12; como 7 < 12, pesquisa-se 7 na relação {1, 3, 4, 5, 6, 8, 10, 11}; para isto compara-se 7 com 5 e, como 7 > 5, pesquisa-se este valor na relação {6, 8, 10, 11}; pesquisa-se na relação {6, 8}; pesquisa-se em {6} e conclui-se que 7 não está relação.

```
function PesqBinaria(var v : TVetor; t : integer; x : real) : integer;
var i, Central : integer;
begin
    i := 1;
    Central := t div 2;
    while (x <> v[Central]) and (i <= t) do
        begin
            if (x < v[Central])
                then
                    t := Central – 1
                else
                    i := Central + 1;
            Central := (i + t) div 2;
        end;
    if (i > Central)
        then
            PesqBinaria := -1
        else
            PesqBinaria := Central;
end;
```

A pesquisa binária também é importante no desenvolvimento da lógica de programação pelo fato de que é uma função que pode ser implementada recursivamente, sem que a implementação recursiva seja menos eficiente do que a não recursiva. Para perceber a recursividade, basta ver que a mesma pesquisa se repete, sendo que, em cada repetição, o vetor pesquisado tem alterado a posição da sua última componente ou da sua primeira componente.

```
function BinRec(var v : TVetor; i, t : integer; x : real) : integer;
var i, c : integer;
begin
    c := (i + t) div 2;
    if  v[c] = x
        then
            BinRec := c
        else
            if i > t
                then
                    BinRec := -1
                else
                    if v[c] > x
                        then
                            BinRec := BinRec(v, i, c-1, x)
                        else
                            Rec := BinRec(v, c + 1, x);
end;
```

9.2 Ordenação

O segundo problema clássico a ser discutido é conhecido como *ordenação* ou *classificação* e foi introduzido na seção 3.5. Este problema consiste em se colocar numa ordem preestabelecida uma relação de valores. Na seção referida, mostramos como ordenar uma relação contendo três valores. Neste capítulo, apresentaremos algoritmos para ordenar uma lista com quaisquer números de valores. A ordenação de uma relação é realizada para que a leitura dos resultados seja facilitada ou para que, como vimos acima, pesquisas sejam realizadas com mais eficiência. Um exemplo prático da necessidade da ordenação ocorre na confecção da lista dos aprovados num concurso vestibular. Algumas universidades divulgam esta lista com os nomes dos aprovados em ordem alfabética e outra em ordem de classificação. Tanto num caso como no outro há necessidade de ordenação. Nesta seção apresentaremos dois algoritmos para ordenação de dados.

9.2.1 O *SelecSort*

O algoritmo *SelectSort* consiste em se selecionar, sucessivamente, o maior elemento, o segundo maior elemento, o terceiro maior elemento etc., e, após cada seleção, armazenar o valor selecionado num vetor auxiliar na posição que mantém o tal vetor auxiliar ordenado. Por exemplo, se se pretende a ordenação em ordem crescente, o "primeiro maior valor" é armazenado na última posição do vetor auxiliar; o "segundo maior valor" é armazenado na penúltima posição do vetor auxiliar e assim sucessivamente. Para que se obtenha o "segundo maior valor" do vetor, excluímos o "primeiro maior valor" atribuindo a esta componente um valor sabidamente menor do que todos os valores armazenados no vetor. Por exemplo, se os valores do vetor são positivos, pode-se atribuir -1 a cada componente já selecionada e já armazenada no vetor auxiliar.

Para exemplificar o método, vamos ordenar o vetor v = {5, 2, 7, 1, 8}. Basta percorrer o vetor 5 vezes selecionando sucessivamente 8, 7, 5, 2 e 1 e realizando as seguintes atribuições:

1. Aux = {, , , , 8}
   v = {5, 2, 7, 1, -1}
2. Aux = { , , , 7, 8}
   v = {5, 2, -1, 1, -1}
3. Aux = { , , 5, 7, 8}
   v = {-1, 2, -1, 1, -1}
4. Aux = { , 2, 5, 7, 8}
   v = {-1, -1, -1, 1, -1}
5. Aux = {1, 2, 5, 7, 8}
   v = {-1, -1, -1, -1, -1},

Para finalizar, basta armazenar nas componentes de *v* as componentes de *Aux*, o que pode ser feito através de uma atribuição "global" v := Aux.

```
{Procedimento que retorna o maior valor de uma relacao e a posicao deste maior valor}
procedure MaiorElemento (var v : TVetor; t : integer; var Maior : integer; var p : integer);
var i : integer;
begin
    Maior := v[1];
    p := 1;
    for i := 1 to t do
        if v[i] > Maior
            then
                begin
                    Maior := v[i];
                    p := i;
                end;
end;
procedure SelectSort(var v : TVetor; t : integer);
var i, j, k : integer;
    Aux : TVetor;
begin
    j := t;
    for i := 1 to t do
        begin
            MaiorElementoPos(v, t, Aux[j], k);
            v[k] := -1;
            j := j - 1;
        end;
    v := Aux;
end;
```

Observe a importância da utilização da segunda versão do procedimento *MaiorElemento* discutido no

exemplo 2 da seção 6.5. Como o parâmetro *Maior* é passado por referência, *o procedimento* já armazena no vetor *Aux* os maiores elementos de v, nas suas posições definitivas.

9.2.2 O *BubbleSort*

O algoritmo *BubbleSort* consiste em se percorrer o vetor a ser ordenado várias vezes, comparando-se cada elemento com o seguinte, permutando suas posições se eles não estiverem na ordem pretendida. Assim, cada vez que o vetor é percorrido, o maior elemento (ou o menor, se a ordenação for feita em ordem decrescente) ainda não ordenado, é colocado na sua posição de ordenação definitiva. Naturalmente, o vetor será percorrido até que não haja mais trocas a se fazer, quando então ele estará ordenado. Por exemplo, se o vetor a ser ordenado em ordem crescente fosse v = {5, 1, 9, 3, 7, 2}, teríamos as seguintes configurações para *v*, de acordo com a ordem de percurso:

Percurso	v
0	{5, 1, 9, 3, 7, 2}
1	{1, 5, 9, 3, 7, 2}
	{1, 5, 3, 9, 7, 2}
	{1, 5, 3, 7, 9, 2}
	{1, 5, 3, 7, 2, 9}
2	{1, 3, 5, 7, 2, 9}
	{1, 3, 5, 2, 7, 9}
3	{1, 3, 2, 5, 7, 9}
4	{1, 2, 3, 5, 7, 9}

```
procedure BubbleSort(var v : TVetor;   t : integer);
var Tr : boolean;
     i, j : integer;
        {Procedimento para permutar os conteudos de duas variaveis}
        procedure Troca(var x, y : integer);
        begin
            x := x + y;
            y := x - y;
            x := x - y;
        end;
begin
    Tr := true;
    while Tr do
        begin
            Tr := false;
            t := t - 1;
            for i := 1 to t do
                if v[i] > v[i+1]
                    then
                        begin
                            Troca(v[i], v[i+1]);
                            Tr := true;
                        end;
        end;
end;
```

Observe que a variável *Tr* verifica se houve alguma troca para que outro percurso seja realizado. Observe também que o comando t := t – 1 se justifica pelo fato de que no percurso de ordem *i*, *i* – 1 elementos já estão em suas posições definitivas.

9.3 Exercícios propostos

1. A maioria das pessoas acha que são azaradas quando procuram uma ficha numa pilha, sempre tendo receio que a ficha procurada seja uma das últimas da pilha. Uma pessoa que acredite ser assim azarada pode pesquisar a tal ficha pesquisando, sucessivamente, a parte superior e a parte inferior da pilha. Assim, verifica a primeira ficha, em seguida, a última, em seguida, a segunda ficha, em seguida, a penúltima e assim sucessivamente. Escreva um procedimento que implemente este método de pesquisa.

2. Uma versão melhorada do *SelectSort* para um vetor que possui i componentes consiste em se comparar a maior dentre as i - 1 primeiras componentes com a última componente, permutando-se suas posições se aquela maior componente for menor do que esta última componente. Desta forma, coloca-se o maior elemento na posição desejada. Este raciocínio é repetido no vetor das i - 1 primeiras componentes e assim sucessivamente. Escreva um procedimento que implemente esta versão do *SelectSort*.

3. O algoritmo *InsertSort* para ordenação de um vetor *Vet* consiste em se tomar um vetor auxiliar *Aux*, contendo uma única componente *Vet[1]* e, em seguida, inserem-se as demais componentes de *Vet*, uma a uma, em *Aux*, de modo que *Aux* se mantenha ordenado, como solicitado no exercício 10 da seção 6.9. Escreva um procedimento que implemente o *InsertSort*.

4. Escreva um procedimento que ordene um arquivo, cujos registros têm os campos *char Cpf[12]* e *char Nome[40]*, em relação ao campo *Cpf*.

Observação

Para receber as respostas dos exercícios propostos, encaminhe mensagem para jaime@ccen.ufal.br, assunto RESPOSTAS EXERCÍCIOS PASCAL, contendo NOME, INSTITUIÇÃO (se for o caso), CIDADE/ESTADO e CATEGORIA (docente, estudante ou auto-didata).

Capítulo 10 Tipos de Dados Definidos pelo Usuário e Conjuntos

10.1 O que são tipos de dados definidos pelo usuário

Além dos tipos de dados predefinidos, os sistemas que implementam Pascal permitem que o programador defina novos tipos de dados, que podem ser usados para facilitar a legibilidade e a manutenção do programa. Infelizmente, a aplicação destes tipos é limitada em função de que variáveis de tipos definidos pelo usuário, com algumas exceções, não podem ser "lidas" nem "escritas". Isto significa, por exemplo, que o armazenamento de valores em variáveis deste tipo só podem ser realizadas através da utilização de comandos de atribuição.

10.2 O tipo *discreto*

Um *tipo discreto* (ou *tipo enumerado*) é um conjunto de identificadores declarado com a seguinte sintaxe:

type Identificador = (Ident1, Ident2,...,IdentN);

Por exemplo, um programa que manipule datas e que pretendesse associar nomes aos dias da semana e aos meses poderia ter definições dos seguintes tipos:

type TDia = (dom, seg, ter, qua, qui, sex, sab);
TMes = (jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez);

A partir daí pode-se declarar variáveis destes tipos, as quais só podem receber atribuições daqueles identificadores. Por exemplo, poderíamos efetuar declarações do tipo:

var Dia : TDia;
Mes : meses;

e atribuições como

Dia := dom;
Mes := fev, etc.

Vale observar que um *tipo discreto* é um tipo ordenado com ordenação definida pela própria colocação dos identificadores na definição do tipo. Sendo ordenado, uma variável de um tipo discreto pode ser argumento das funções *Succ* e *Pred* definidas na seção 2.10 e podem ser operandos de operadores relacionais. No exemplo anterior, temos *Succ*(seg) = ter, *Pred*(jul) = jun, seg < ter, nov > abr, etc.. É claro que nem todo valor da variável pode ser argumento daquelas funções: sab não pode ser argumento de *Succ* nem jan pode ser argumento de *Pred*.

Um tipo discreto também pode ser usado como limites de um comando *for* ou do conjunto de índices de um vetor. Por exemplo, poderíamos, ainda com o exemplo anterior, ter uma declaração do tipo

var Faturamento : **array**[dom..sab] **of real**;

e um comando do tipo

for Mes := mar **to** ago **do**;

10.3 O tipo *faixa*

Outro tipo de dado que pode ser definido pelo usuário é chamado *tipo faixa*, cuja definição é feita através da seguinte sintaxe:

type Identificador = $v_i..v_f$;.

onde v_i e v_f são constantes de um tipo de dado ordenado, chamado *tipo básico*, com $v_i < v_f$. Como um tipo discreto é um tipo ordenado, ele pode ser tipo básico de um tipo faixa. Assim, com o exemplo anterior, poderíamos ter os seguintes tipos:

```
type TDiasUteis = seg..sex
     TInverno = jun..ago;
```

Neste caso, se declarássemos uma variável *DiaDeTrabalho* do tipo *TDiasUteis*, poderíamos atribuir-lhe qualquer um dos valores *seg*, *ter*, *qua*, *qui*, *sex* ou *sab*.

Se o tipo básico é *integer* ou *char*, uma variável de um tipo faixa pode ser "lida" através do comando *readln* e "escrita" através do comando *write*. Na hipótese de se pretender que a execução de um comando *readln* com a entrada de um valor menor que v_i ou maior que v_2 não seja aceita, deve-se ativar a diretiva de compilação {$R+} que faz com que, durante a execução do programa, seja verificado se cada dado de entrada pertence ao intervalo definidor do tipo faixa. Com esta diretiva ativada, qualquer entrada de dado não pertencente à faixa fixada provoca um *erro de execução* e a execução do programa é abortada.

Um tipo faixa pode ser usado diretamente na declaração de um vetor, escrevendo-se, nos colchetes, o identificador do tipo no lugar dos limites inferior e superior. Assim, mais uma vez com o exemplo acima, poderíamos declarar:

```
var Faturamento : array[TDiasUteis] of real;.
```

Outro exemplo:

```
type TDig = 0..9;
var v : array[TDig] of integer;.
```

A título de exemplo, o programa abaixo computa as médias dos números de trabalhos publicados pelos professores lotados num Centro de Ciências Exatas e Naturais, dados os números de trabalhos publicados pelos professores de cada um dos departamentos que compõem o referido Centro.

```
program ProducaoCientifica;
type TDepartamento = (Est, Fis, Geo, Inf, Mat, Qui);
var Dep : TDepartamento;
    Trab, Prof : array[Est..Qui] of integer;
    MediaDep : array[Est..Qui] of real;
    MediaCentro : real;
    NumTrab, NumProf : integer;
begin
   NumTrab := 0; NumProf := 0;
   writeln('Digite o número de professores e o número de trabalhos  publicados de cada
Departamento,  na ordem  Estatística, Física, Geografia, Informática, Matemática e Química');
   for Dep := Est to Qui do
      begin
         readln(Prof[Dep], Trab[Dep]);
         MediaDep[Dep] := Trab[Dep]/Prof[Dep];
         NumProf := NumProf + Prof[Dep];
         NumTrab := NumTrab + Trab[Dep];
      end;
   MediaCentro := NumTrab/NumProf;
   writeln('Media por departamento:');
   writeln('Est  Fis  Geo  Inf  Mat   Met   Qui');
   for Dep := Est to Qui do
      write(MediaDep[Dep]:4:2,'  ');
   writeln;
   writeln('Media do Centro: ', MediaCentro:4:2);
end.
```

Muitas pessoas questionam as vantagens do uso de tipos definidos pelo usuário principalmente pelo fato já citado de que variáveis deste tipo não podem ser lidas nem escritas. Evidentemente, estes questionamentos são procedentes e, para exemplificar a validade deles, basta observar a necessidade que tivemos de exibir as

siglas dos departamentos como mensagens, em vez de conteúdos da variável *Dep*, o que seria mais conveniente. No exemplo, talvez fosse preferível armazenar as siglas dos departamentos num vetor de *strings*, indexado por inteiros associados aos departamentos.

10.4 Conjuntos

A linguagem Pascal possui um tipo de dado pré-definido capaz de armazenar, sem indexações, elementos de um conjunto, permitindo, com variáveis deste tipo, a realização de *uniões*, *interseções* e *diferenças* como definido na matemática, e a verificação do fato de que um dado elemento está ou não armazenado na variável.

A definição deste tipo de dado, normalmente chamado de *conjunto*, é feita através da seguinte sintaxe:

var Identificador : **set of** tipo de dado;.

onde *tipo de dado*, denominado *tipo base* do conjunto, define que elementos podem ser armazenados na variável, podendo ser qualquer tipo ordenado com até 255 elementos.

Por exemplo, um programa para a realização de um bingo eletrônico poderia ter definições do seguinte tipo:

```
type TPedras = 1..75;
var Urna : set of pedras;
```

Do mesmo modo que variáveis de tipos definidos pelo usuário, variáveis do tipo *set* não podem ser "lidas" nem "escritas". O armazenamento de elementos numa variável do tipo *set* só pode ser feito, portanto, através de um comando de atribuição, podendo ser utilizada uma das seguintes formas:

1. Enumeração explícita dos elementos entre colchetes:

   Conj := [1, 2, 3, 4, 5, 6, 7, 8, 9];

2. Especificação, entre colchetes, de uma faixa do tipo base:

   Conj := [1..9];

3. Combinação das formas anteriores:

   OutroConj := [1..9, 21..30, 41, 43, 45, 47, 49];.

O conjunto vazio também pode ser "armazenado" numa variável do tipo *set*. Para isto, pode-se declarar uma variável *Vazio* do tipo *set* e realizar a seguinte atribuição:

Vazio := [ ];

Como veremos em exemplos a seguir, o conjunto vazio é utilizado para a inicialização de conjuntos e para interrupções de estruturas de repetição.

Como já foi observado, podem-se realizar operações matemáticas com variáveis ou constantes de tipo *set*. Estas operações são realizadas através dos operadores **+** (união), ***** (interseção) e **-** (diferença).

O sistema também oferece o operador *in* que verifica se um elemento do tipo base está armazenado no conjunto. Este operador é muito utilizado para que o programa se proteja de entradas de dados indesejáveis. Por exemplo, se uma variável *Letra*, do tipo *char* deve receber uma letra, pode-se utilizar uma estrutura como a seguinte:

```
repeat
    readln(Letra);
until Letra in ['a'..'z', 'A'..'Z'];
```

O operador *in* também é utilizado em expressões lógicas de comandos *if* quando o número de valores possíveis para a variável que definirá a decisão não é pequeno. Por exemplo, pretendendo-se verificar se uma variável *Letra* é uma vogal, pode-se usar:

```
if UpCase(LETRA) in ['A', 'E', 'I', 'O', 'U'];
    then ...
```

10.5 Exemplos Parte XIII

1. O exemplo a seguir apresenta um método clássico da obtenção dos números primos menores que um inteiro positivo *n*, dado. Este método, conhecido como *Crivo de Eratóstenes*, consiste, para n = 300, em escrever o número 2 e todos os inteiros ímpares maiores do que 1 e menores que 300 e riscar os múltiplos de 3, de 5 e de 7. A matemática prova que os números não-riscados são todos primos:

2	3	5	7	~~9~~	11	13	~~15~~	17	19
~~21~~	23	~~25~~	~~27~~	29	31	~~33~~	~~35~~	37	~~39~~
41	43	~~45~~	47	~~49~~	~~51~~	53	~~55~~	~~57~~	59
61	~~63~~	~~65~~	67	~~69~~	71	73	~~75~~	~~77~~	79
~~81~~	83	~~85~~	~~87~~	89	~~91~~	~~93~~	~~95~~	97	~~99~~
101	103	~~105~~	107	109	~~111~~	113	~~115~~	~~117~~	~~119~~
~~121~~	~~123~~	~~125~~	127	~~129~~	131	~~133~~	~~135~~	137	139
~~141~~	~~143~~	~~145~~	~~147~~	149	151	~~153~~	~~155~~	157	~~159~~
~~161~~	163	~~165~~	167	~~169~~	~~171~~	173	~~175~~	~~177~~	179
181	~~183~~	~~185~~	~~187~~	~~189~~	191	193	~~195~~	197	199
~~201~~	~~203~~	~~205~~	~~207~~	~~209~~	211	~~213~~	~~215~~	~~217~~	~~219~~
~~221~~	223	~~225~~	227	229	~~231~~	233	~~235~~	~~237~~	239
241	~~243~~	~~245~~	~~247~~	~~249~~	251	~~253~~	~~255~~		

O programa seguinte utiliza dois conjuntos: um contendo todos os ímpares menores que *n* e o outro, inicializado com o conjunto [2], que vai recebendo, sucessivamente, cada menor elemento do conjunto *Crivo*.

```
program CrivoDeEratostenes;
type TConjunto = set of 1..255;
var Crivo, Primos : TConjunto;
     i, Primo, n : integer;
begin
    writeln('Digite um inteiro menor que 255');
    readln(n);
    Crivo := [];
    for i := 3 to n do
        if i mod 2 = 1
            then
                Crivo := Crivo + [i];
    Primos := [2];
    Primo := 3;
    while Crivo <> [ ] do
        begin
            while not (Primo in Crivo) do
                Primo := Primo + 2;
            Primos := Primos + [Primo];
            i := Primo;
            while i <= n do
                begin
                    Crivo := Crivo - [i];
                    i := i + Primo;
                end;
        end;
```

```
    writeln('Primos menores que :', n);
    for i := 2 to n do
        if i in Primos
            then write(I,'  ');
end.
```

Observe que, como o comando *writeln* não aceita argumentos do tipo *set* nem se tem a possibilidade de acesso a cada elemento do conjunto, a exibição na tela dos elementos de um conjunto se faz percorrendo todos os elementos do tipo base do conjunto, verificando com o operador *in* se tal elemento do tipo base é elemento do conjunto. Naturalmente, esta forma de exibir um conjunto também limita sobremaneira a utilização do tipo *set*.

2. Aproveitamos o próximo exemplo para apresentar a função predefinida *Random*(x), que retorna um inteiro aleatório i, com $0 \le i < x$ e que também pode ser ativada sem parâmetros, caso em que retornará um número real r, com $0 \le r < 1$. Em qualquer dos casos, a ativação da função deve ser precedida da chamada do procedimento *Randomize*, que permite a geração do número aleatório.

O objetivo do exemplo é um programa que permita sorteio eletrônico de um bingo. Para isto,ele utiliza um conjunto *Urna* contendo as "pedras" de 1 (um) a 75 (setenta e cinco), sendo o sorteio realizado pela função *Random* dentro de uma estrutura de repetição para que uma pedra não seja sorteada mais de uma vez. Quando o sorteio é realizado, a "pedra" sorteada é retirada da urna, utilizando-se para isto a diferença entre conjuntos.

```
program Bingo;
type TPedras = 1..75;
var Urna : set of TPedras;
     NumPedra, Pedra : integer;
     Bateu : char;
begin
    Urna := [1..75];
    NumPedra := 0; Bateu := 'N';
    Randomize;
    writeln('Pedras sorteadas : ');
    while (Urna <> [])  and (Bateu <> 'S' ) do
        begin
            writeln('Alguem bateu? (S/N)');
            readln(Bateu);
            Bateu := UpCase(Bateu);
            if Bateu <> 'S'
                then
                    begin
                        repeat
                            Pedra := Random(75) + 1;
                        until Pedra in Urna;
                        writeln('Pedra sorteada: ', Pedra );
                        NumPedra := NumPedra + 1;
                        writeln('Numero de pedras: ', NumPedra);
                        Urna := Urna - [Pedra];
                    end;
        end;
end.
```

Capítulo 11 Alocação Dinâmica da Memória

11.1 O que é alocação dinâmica: ponteiros

Até agora, os programas utilizavam a memória do computador *estaticamente*: todas as posições de memória eram reservadas para as variáveis no início da execução do programa ou do subprograma e, mesmo que não tivessem sendo mais utilizadas, continuavam reservadas para as mesmas variáveis até a conclusão da execução do programa ou do subprograma. Um vetor global do tipo *real* com dez mil componentes, por exemplo, "ocupará" quarenta mil bytes de memória durante toda a execução do programa. Naturalmente, isto pode, em grandes programas, sobrecarregar ou até mesmo esgotar a memória disponível. No primeiro caso, há uma degradação na eficiência do programa; no segundo caso, a execução do programa pode ser inviabilizada.

A linguagem Pascal permite a *alocação dinâmica* da memória de tal modo que posições de memória sejam reservadas para variáveis no instante em que sejam necessárias e sejam liberadas (as posições de memória) para o sistema nos instantes em que não estejam sendo utilizadas. Para compreender como isto é feito, estudaremos os *ponteiros*.

Um *ponteiro* (*pointer*) é uma variável capaz de armazenar endereços de outras variáveis, devendo ser definido através da sintaxe

var Identificador : ^ tipo de dado;

onde *tipo de dado* indicará o tipo de dado de uma variável cujo endereço pode ser armazenado na variável *Identificador*.

Como em outros casos, e isto é o mais comum, pode-se definir um tipo ponteiro através da sintaxe óbvia

type Identificador = ^ tipo de dado;

sendo que, a partir daí, qualquer variável do tipo *Identificador* pode armazenar o endereço de uma variável do tipo de dado do segundo membro da declaração. É importante observar que, se o tipo de dado for um tipo estruturado, é necessário associar-lhe um identificador de tipo, para que este identificador de tipo seja utilizado na definição do ponteiro.

Quando se define um tipo ponteiro associado a um tipo de dado não-predefinido, o sistema procura em toda área de definições de tipos pela definição daquele tipo. Isto significa que o tipo de dado associado ao ponteiro pode ser definido posteriormente. Esta possibilidade é extremamente necessária quando o tipo de dado associado ao ponteiro deve conter o próprio ponteiro como uma de suas componentes, como veremos em exemplo seguinte.

Pascal possui uma constante predefinida do tipo ponteiro *nil* que não está associado a nenhum endereço. Esta constante pode ser atribuída a qualquer variável de tipo ponteiro, sendo o ponteiro também de qualquer tipo. Normalmente, esta constante é utilizada para inicializar variáveis e para encerramento de estruturas de repetição, sendo comum se utilizar a representação gráfica abaixo para representá-la.

A alocação dinâmica da memória é feita pelo procedimento predefinido *New* e a referência à variável *apontada* por um ponteiro é feita por *Identificador^*. Além disto, o procedimento *Dispose* libera para o sistema operacional as posições de memória alocadas para o ponteiro passado para ele. Por exemplo, o programa

```
type TVetor = array[1..100] of integer;
var p : ^TVetor;
    i : integer;
begin
   New(p);
   for i := 1 to 100 do
      p^[i] := i*i;
```

```
    for i := 1 to 100 do
        write(p^[i], ' ');
    Dispose(p);
end.
```

cria dinamicamente um vetor e nele armazena os quadrados dos 100 primeiros inteiros.

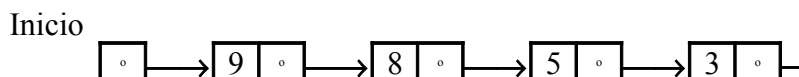
Naturalmente, o leitor pode estar pensando que o programa acima não teria muita vantagem em relação ao programa abaixo, onde o vetor é criado estaticamente,

```
type TVetor = array[1..100] of integer;
var v : TVetor;
i : integer;
begin
    for i := 1 to 100 do
        v[i] := i*i;
    for i := 1 to 100 do
        write(v[i], ' ');
end.
```

De fato, os dois programas, durante suas execuções, utilizam oitenta bytes de memória. Haverá diferença se estes programas fizerem parte de um programa maior. Neste caso, o primeiro utiliza oitenta bytes apenas até a execução do comando *Dispose(p)*, enquanto que o segundo utiliza os oitenta bytes durante toda a execução do programa.

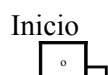
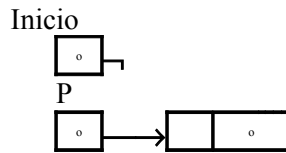
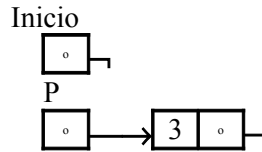
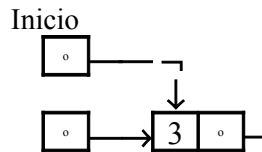
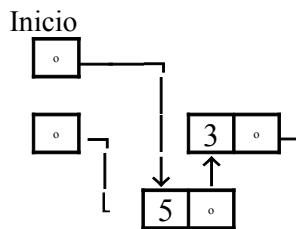
11.2 Listas

Para exemplificar a aplicação de ponteiros, apresentaremos a mais simples estrutura de dados cuja implementação através de alocação dinâmica é muito útil. Trata-se das *listas simplesmente encadeadas* que podem ser definidas como uma seqüência de elementos ligados através de ponteiros, sendo o número máximo de elementos da lista não fixado *a priori*. Usualmente, cada elemento da lista é uma variável do tipo *record*, com um campo contendo um ponteiro e os demais campos contendo os dados que o programa vai manipular. O primeiro elemento é associado a alguma variável do tipo *ponteiro* e o campo do tipo *pointer* do último elemento da lista tem conteúdo *nil*. Isto é indispensável para que se possua referência sobre o início e o final da lista.

Uma lista de inteiros, por exemplo, poderia ser representada graficamente da seguinte forma:

Para se criar a lista do exemplo anterior, necessitamos das seguintes definições de tipos e declarações de variáveis:

```
type TPonteiro = ^TElemento;
     TElemento = record
          Dado : integer;
          Prox : TPonteiro;
     end;
var Inicio, p : TPonteiro;
```

Com estas definições e declarações, realizamos as seguintes etapas:

1. Para que se tenha referência ao primeiro elemento da lista e como no início a lista está vazia, fazemos Inicio := **nil**.

2. Para dar entrada no primeiro *Dado*, criamos um registro referenciado por p^, através da chamada do procedimento *New*(p).

3. Agora damos entrada no primeiro *Dado*, com *readln*(p^.Dado), e atribuímos p^.Prox := Inicio, cujo valor corrente é *nil*.

4. Em seguida, fazemos Inicio := p, para que *Inicio* referencie sempre o início da lista.

5. Para dar entrada em outro *Dado*, criamos um novo registro com *New*(p), lemos este outro *Dado* com *readln*(p^.Dado) e atualizamos p^.Prox := Inicio e Inicio := p.

Evidentemente as etapas de índices 2 a 5 devem estar numa estrutura de repetição, permitindo que se forneçam todos os elementos da lista. A etapa 5 indica que o início da lista, o elemento referenciado por *Inicio*, é sempre o último elemento a dar entrada.

O procedimento seguinte traduz para o Pascal as idéias discutidas acima.

```
procedure CriaLista;
begin
    Inicio := nil;
    writeln('Digite os números ( -1 para encerrar)');
    repeat
        New(p);
        readln(p^.Dado);
        if p^.Dado <> -1
            then
                begin
                    p^.Prox := Inicio;
                    Inicio := p;
                end;
    until p^.Dado = -1;
end;
```

Para se realizar uma pesquisa nesta lista, inicializamos *p* com o ponteiro *Inicio* (pois este aponta para o início da lista) e percorremos a lista até encontrar o elemento ou até que o valor de *p* seja *nil*, o que acontecerá quando o elemento pesquisado não for elemento da lista.

```
function Pesquisa(x : integer; var p, Anterior : TPonteiro) :  boolean;
var  Achou : boolean;
begin
```

```
        Achou := false;
        p := Inicio;
        Anterior := Inicio;
        while (p <> nil) and not Achou do
            if p^.Dado = x
                then
                    Achou := true
                else
                    begin
                        Anterior := p;
                        p := p^.Prox;
                    end;
        Pesquisa := Achou;
    end;
```

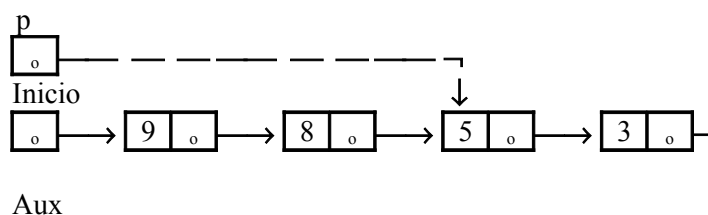
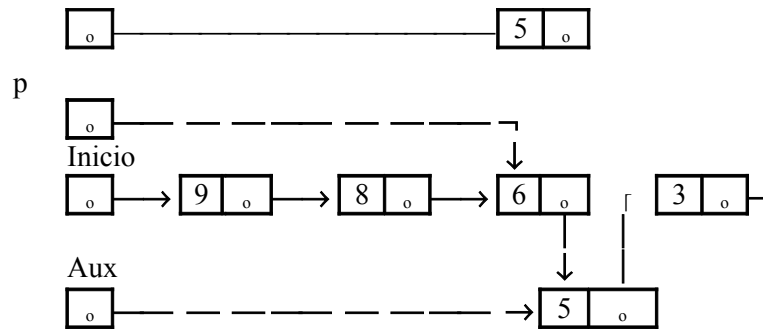
Note que o procedimento retorna tanto o ponteiro que está apontando para o elemento (quando a busca é bem sucedida), como o ponteiro que aponta para o elemento anterior. Este ponteiro, *Anterior*, é útil para se efetuarem inserções e deleções em listas ordenadas.

O processamento de uma lista se faz de maneira natural. Fazemos *p* apontar para o início da lista e a percorremos realizando o processamento desejado até que não haja mais elementos, o que acontecerá quando o conteúdo de *p* for *nil*. Para exemplificar, apresentamos o procedimento abaixo que exibe todos os elementos da lista.

```
procedure EscreveLista;
begin
    p := Inicio;
    while p <> nil do
        begin
            write(p^.Dado,'  ');
            p := p^.Prox;
        end;
end;
```

A inserção de um elemento numa lista ordenada requer um procedimento que determine a posição de inserção, procedimento semelhante à função *Pesquisa* discutido acima.

```
procedure Posicao(var x : integer; var p, Anterior :  TPonteiro);
begin
    p := Inicio;
    while (p <> nil) and (p^.Dado > x) do
        begin
            Anterior := p;
            p := p^.Prox;
        end;
end;
```

Se a posição de inserção for anterior ao último elemento da lista, a inserção é feita antes do elemento apontado pelo ponteiro *p* obtido pelo procedimento *Posicao*. Neste caso, transfere-se o registro apontado por *p* para um ponteiro auxiliar *Aux*, faz-se o ponteiro deste elemento apontar para o registro apontado por *Aux* e armazena-se no campo denominado *Dado* do elemento apontado por *p* o elemento que se quer inserir. A figura a seguir indica graficamente estas operações se quiséssemos inserir x = 6.

Se a inserção deve ser feita no final lista, *p* terá conteúdo *nil* e então o elemento deve ser inserido após o elemento apontado pelo ponteiro *Anterior* obtido também pelo procedimento *Posicao*. Neste caso, cria-se um novo elemento apontado por *p*, atribui-se o valor a ser inserido no campo *Dado* deste elemento e aponta-se o ponteiro deste elemento para o elemento apontado pelo campo ponteiro de *Anterior*.

Estas idéias estão implementadas no procedimento a seguir.

```
procedure InsereLista(x : integer);
var Aux, p, Anterior : TPonteiro;
begin
    Posicao(x, p, Anterior);
    if p <> nil
        then
            begin
                Aux^.Prox := p^.Prox;
                p^.Prox := Aux;
                Aux^.Dado := p^.Dado;
                p^.Dado := x;
            end
        else
            begin
                New(p);
                p^.Dado  := x;
                p^.Prox := Anterior^.Prox;
                Anterior^.Prox := p;
            end;
end;
```

Para finalizar, discutiremos a remoção de um elemento da lista. Se o elemento a remover é o primeiro elemento da lista, basta atualizar o ponteiro *Inicio*, apontando-o para o segundo elemento da lista. Se o elemento a remover não é o primeiro da lista, basta apontar o ponteiro do elemento apontado por *Anterior* para o elemento seguinte àquele apontado por *p* e liberar a variável apontada por *p* através do procedimento predefinido *Dispose*.

```
procedure Remove(var x : integer);
var Aux , p, Anterior : TPonteiro;
begin
    if Pesquisa(x, p, Anterior)
        then
            begin
                if (Anterior = Inicio) and (p = Inicio)
                    then
                        Inicio := p^.Prox
                    else
                        Anterior^.Prox := p^.Prox;
                Dispose(p);
            end;
end;
```

Bibliografia

Dijkstra, E. W., *A Discipline of Programiming*. Prentice-Hall. New Jersey, 1975.

Evaristo, J. e Crespo, S, *Aprendendo a Programar Programando numa Linguagem Algorítmica Executável (ILA)*. Book Express, Rio de Janeiro, 2000.

Evaristo, J., *Aprendendo a Programar Programando em Linguagem C*. Book Express, Rio de Janeiro, 2001.

Evaristo, J., *Aprendendo a Programar Programando em Turbo Pascal*. Editora da Universidade Federal de Alagoas (EDUFAL). Alagoas, 1996.

Evaristo, J., *Introdução à Algebra (com aplicações à Ciência da Computação*). Editora da Universidade Federal de Alagoas (EDUFAL). Alagoas, 1999.

Farrer, Harry e outros, *Pascal Estruturado*. Editora Guanabara. Rio de Janeiro, 1985.

Grogono, Peter, *Programming in Pascal*. Addison-Wesley. Massachusetts, 1985.

Knuth, D. E., *The Art of Computer Programming*, volume 2, *Seminumerical Algorithms*, Addison-Wesley Publishing Company. USA, 1988.

Kowaltowski, T. & Lucchesi, C., *Conceitos Fundamentais e Teoria da Computação*. Anais do II WEI. Minas Gerais, 1994

Mecler, Ian & Maia, Luiz Paulo, *Programação e Lógica com Turbo Pascal*. Editora Campus. Rio de Janeiro, 1989.

Norton, P., *Introdução à Informática*. Makron Books. Sãp Paulo, 1996.

Rangel, J. L., *Os Programas de Graduação em Linguagens de Programação*. Anais do II WEI. Minas Gerais, 1994.

Schmitz, Eber Assis & Teles, A. S. de Souza, *Pascal e Técnicas de Programação*. LTC Editora. Rio de Janeiro, 1988.

Szwarcfiter, J. L. & Markenzon, *Estruturas de Dados e seus Algoritmos*. LTC Editora. Rio de Janeiro, 1994.

Wirth, N., *Algorithms & Data Structures*. Prentice-Hall. New-Jersey, 1986.

Índice remissivo

R

S

Aprenda os conceitos e fundamentos da programação em Pascal, uma linguagem padrão de mercado, utilizada em diversas ferramentas de desenvolvimento, incluindo o Delphi

O objetivo deste livro é o desenvolvimento da lógica de programação; ele pode ser utilizado por quem está iniciando o apredizado de produtos como Delphi, ou ainda como livro texto para a primeira disciplina de programação de computadores dos cursos de Ciência da Computação, Sistema de Informação, Engenharia da Computação, e para a programação das disciplinas do tipo Introdução à Computação, dos cursos da área de Ciências Exatas.

Além disso, este livro é útil como apoio ao ensino específico da linguagem Pascal, base para o estudo do ambiente de programação Delphi. Pela sua concepção, pode também ser utilizado para a aprendizagem autodidática de Pascal.

O **Prof. Jaime Evaristo** formou-se em Engenharia Civil em 1973 na antiga Faculdade de Engenharia da Universidade Federal de Alagoas, mas nunca exerceu a profissão de engenheiro. Nos primeiros anos de sua graduação já havia optado pela carreira de magistério, tendo lecionado em diversos colégios e cursos preparatórios para o vestibular de Maceió. Em 1974, ingressou no quadro de professores da Universidade Federal de Alagoas, tendo, no ano seguinte, realizado o Curso de Mestrado em Matemática na Universidade Federal de Pernambuco. É autor de livros consagrados, como *Aprendenda Programando em Linguagem C.*

WWW.BOOKEXPRESS.COM.BR
(0xx21) 2560-8980 / 2560-8018 Fax: 2561-0838
Email: sac@bookexpress.com.br

ISBN 857584004-5